EFFECTIVE C

AN INTRODUCTION TO
PROFESSIONAL C PROGRAMMING

ROBERT C. SEACORD

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

Welcome to the Early Access edition of the as yet unpublished *Effective C: An Introduction to Professional C Programming* by Robert C. Seacord! As a prepublication title, this book may be incomplete and some chapters may not have been proofread.

Our goal is always to make the best books possible, and we look forward to hearing your thoughts. If you have any comments or questions, email us at **earlyaccess@nostarch.com**. If you have specific feedback for us, please include the page number, book title, and edition date in your note, and we'll be sure to review it. We appreciate your help and support!

We'll email you as new chapters become available. In the meantime, enjoy!

EFFECTIVE C

ROBERT C. SEACORD

Early Access edition, 5/7/20

ISBN-10: 1-7185-0104-8
ISBN-13: 978-1-71850-104-1

Publisher: William Pollock
Production Editor: Katrina Taylor
Cover Illustration: Gina Redman
Developmental Editors: Liz Chadwick and Frances Saux
Technical Reviewer: Martin Sebor
Copyeditor: Sharon Wilkey
Compositor: Happenstance Type-O-Rama
Proofreader: Emelie Battaglia

CONTENTS

INTRODUCTION

C was developed as a system programming language in the 1970s, and even after all this time, it remains incredibly popular.

System languages are designed for performance and ease of access to the underlying hardware while providing high-level programming features. While other languages may offer newer language features, their compilers and libraries are typically written in C. Carl Sagan once said, "If you wish to make an apple pie from scratch, you must first invent the universe."

The inventors of C did not invent the universe; they designed C to work with a variety of computing hardware and architectures that, in turn, were constrained by physics and mathematics. C is layered directly on top of computing hardware, making it more sensitive to evolving hardware features, such as vectorized instructions, than higher-level languages that typically rely on C for their efficiency.

According to the TIOBE index, C has been either the most popular programming language or second most popular since 2001.[1] C is TIOBE's programming language of the year for 2019. The popularity of the C programming language can most likely be attributed to several tenets of the language referred to as the *spirit of C*:

- Trust the programmer. Generally speaking, the C language assumes you know what you're doing and lets you. This isn't always a good thing (for example, if you don't know what you're doing).
- Don't prevent the programmer from doing what needs to be done. Because C is a system programming language, it has to be able to handle a variety of low-level tasks.
- Keep the language small and simple. The language is designed to be fairly close to the hardware and to have a small footprint.
- Provide only one way to do an operation. Also known as *conservation of mechanism*, the C language tries to limit the introduction of duplicate mechanisms.
- Make it fast, even if it isn't guaranteed to be portable. Allowing you to write optimally efficient code is the top priority. The responsibility of ensuring that code is portable, safe, and secure is delegated to you, the programmer.

A Brief History of C

The C programming language was developed in 1972 by Dennis Ritchie and Ken Thompson at Bell Telephone Laboratories. Brian Kernighan coauthored *The C Programming Language* (K&R 1988) with Dennis Ritchie. In 1983, the American National Standards Institute (ANSI) formed the X3J11 committee to establish a standard C specification, and in 1989, the C Standard was ratified as ANSI X3.159-1989, "Programming Language C." This 1989 version of the language is referred to as *ANSI C* or *C89*.

In 1990, the ANSI C Standard was adopted (unchanged) by a joint technical committee of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) and published as the first edition of the C Standard, C90 (ISO/IEC 9899:1990). The second edition of the C Standard, C99, was published in 1999 (ISO/IEC 9899:1999), and a third edition, C11, in 2011 (ISO/IEC 9899:2011). The latest version of the C Standard (as of this writing) is the fourth version, published in 2018 as C17 (ISO/IEC 9899:2018). A new major revision referred to as C2x is under

1. The TIOBE Programming Community index is an indicator of the popularity of programming languages and can be found at *https://www.tiobe.com/tiobe-index/*. The ratings are a measure of the number of skilled engineers, courses, and third party vendors for each language. The index can be used to help you decide which programming languages to learn or adopt when building a new software system.

development by ISO/IEC. According to 2018 polling data from JetBrains, 52 percent of C programmers use C99, 36 percent use C11, and 23 percent use an embedded version of C.[2]

The C Standard

The *C Standard* (ISO/IEC 9899:2018) defines the language and is the final authority on language behavior. While the standard can be obscure to impenetrable, you need to understand it if you intend to write code that's portable, safe, and secure. The C Standard provides a substantial degree of latitude to implementations to allow them to be optimally efficient on various hardware platforms. *Implementations* is the term used by the C Standard to refer to compilers and is defined as follows:

> A particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment.

This definition indicates that each compiler with a particular set of command line flags, along with the C Standard Library, is considered a separate implementation, and different implementations can have significantly different *implementation-defined behavior*. This is noticeable in GNU Compiler Collection (GCC), which uses the `-std=` flag to determine the language standard. Possible values for this option include `c89`, `c90`, `c99`, `c11`, `c17`, `c18`, and `c2x`. The default depends on the version of the compiler. If no C language dialect options are given, the default for GCC 10 is `-std=gnu17`, which provides extensions to the C language. For portability, specify the standard you're using. For access to new language features, specify a recent standard. A good choice (in 2019) with GCC 8 and later is `-std=c17`.

Because implementations have such a range of behaviors, and because some of these behaviors are undefined, you can't understand the C language by just writing simple test programs to examine the behavior.[3] The behavior of the code may vary when compiled by a different implementation on different platforms or even the same implementation using a different set of flags or a different C Standard Library implementation. Code behavior can even vary between *versions* of a compiler. The C Standard is the only document that specifies which behaviors are guaranteed for all implementations, and where you need to plan for variability. This is mostly a concern when developing portable code, but can also affect the security and safety of your code.

2. See the JetBrains website for more details on the survey: *https://www.jetbrains.com/lp/devecosystem-2019/c/*.

3. If you want to try this, the Compiler Explorer is an excellent tool; see *https://godbolt.org/*.

The CERT C Coding Standard

The CERT® C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems (Seacord 2014) is a reference book I wrote while managing the secure coding team at the Software Engineering Institute at Carnegie Mellon University. *The CERT C Coding Standard* contains examples of common C programming mistakes and how to correct them. Throughout this book, we reference some of these rules as a source for detailed information on specific C language programming topics.

Who This Book Is For

This book is an introduction to the C language. It is written to be as accessible as possible to anyone who wants to learn C programming, without dumbing it down. In other words, we didn't overly simplify C programming in the way many other introductory books and courses might. These overly simplified references will teach you how to get code to compile and run, but the code might still be wrong. Developers who learn how to program C from such sources will typically develop substandard, flawed, insecure code that will eventually need to be rewritten (often sooner than later). Hopefully, these developers will eventually benefit from senior developers in their organizations who will help them unlearn these harmful misconceptions about programming in C, and help them start developing professional quality C code. On the other hand, this book will quickly teach you how to develop correct, portable, professional-quality code, build a foundation for developing security-critical and safety-critical systems, and perhaps teach you a thing or two that even the senior developers at your organization don't know.

Effective C: An Introduction to Professional C Programming is a concise introduction to essential C language programming that will have you writing programs, solving problems, and building working systems in no time. The code examples are idiomatic and straightforward.

In this book, you'll learn about essential programming concepts in C and practice writing high-quality code with exercises for each topic. You'll also learn about good software engineering practices for developing correct, secure C code. Go to this book's page at *https://www.nostarch.com/effective_c/* or to *http://www.robertseacord.com/* where we will provide updates and additional material. When you have completed this book, if you are interested in learning more about secure coding in C, C++, or other languages, please check out the training classes offered through NCC Group at *https://www.nccgroup.trust/us/our-services/cyber-security/security-training/secure-coding/*.

What's in This Book

This book starts with an introductory chapter that covers just enough material to get you programming right from the start. After this, we circle back and examine the basic building blocks of the language. The book culminates with two chapters that will show you how to compose real-world systems from these basic building blocks and how to debug, test, and analyze the code you've written. The chapters are as follows:

Chapter 1: Getting Started with C You'll write a simple C program to become familiar with using the `main` function. You'll also look at a few options for editors and compilers.

Chapter 2: Objects, Functions, and Types This chapter explores basics like declaring variables and functions. You'll also look into the principles of using basic types.

Chapter 3: Arithmetic Types You'll learn about the two kinds of arithmetic data types: integers and floating-point types.

Chapter 4: Expressions and Operators You'll learn about operators and how to write simple expressions to perform operations on various object types.

Chapter 5: Control Flow You'll learn how to control the order in which individual statements are evaluated. We'll start by going over expression statements and compound statements that define the work to be performed. We'll then cover three kinds of statements that determine which code blocks are executed, and in what order: selection, iteration, and jump statements.

Chapter 6: Dynamically Allocated Memory You'll learn about dynamically allocated memory, which is allocated from the heap at runtime. Dynamically allocated memory is useful when the exact storage requirements for a program are unknown before runtime.

Chapter 7: Character and Strings You'll learn about the various character sets, including ASCII and Unicode, that can be used to compose strings. You'll learn how strings are represented and manipulated using the legacy functions from the C Standard Library, the bounds-checking interfaces, and POSIX and Windows APIs.

Chapter 8: Input/Output This chapter will teach you how to perform input/output (I/O) operations to read data from, or write data to, terminals and filesystems. I/O involves all the ways information enters or exits a program, without which your programs would be useless. We'll cover techniques that make use of C Standard streams and POSIX file descriptors.

Chapter 9: Preprocessor You'll learn how to use the preprocessor to include files, define object- and function-like macros, and conditionally include code based on implementation-specific features.

Chapter 10: Program Structure You'll learn how to structure your program into multiple translation units consisting of both source and include files. You'll also learn how to link multiple object files together to create libraries and executable files.

Chapter 11: Debugging, Testing, and Analysis This chapter describes tools and techniques for producing correct programs, including compile-time and runtime assertions, debugging, testing, static analysis, and dynamic analysis. The chapter also discusses which compiler flags are recommended for use in different phases of the software development process.

You're about to embark on a journey from which you will emerge a newly minted but professional C developer.

1

GETTING STARTED WITH C

In this chapter, you'll develop your first C program: the traditional "Hello, world!" program. I'll take you through the various aspects of this simple program, as well as compiling and running a C program. Then I'll discuss some editor and compiler options, and lay out common portability issues you'll quickly become familiar with as you code in C.

Developing Your First C Program

The best way to learn C programming is to start writing C programs, and the traditional program to start with is "Hello, world!"

To write this program, you need a text editor or *integrated development environment (IDE)*. There are many to choose from, but for now, open your favorite editor, and we'll survey other options later in this chapter.

In your text editor, enter the program in Listing 1-1.

```
#include <stdio.h>
#include <stdlib.h>
❶ int main(void) {
  ❷ puts("Hello, world!");
  ❸ return EXIT_SUCCESS;
❹ }
```

Listing 1-1: The hello.c *program*

We'll go over each line of this program in more detail shortly. For now, save this file as *hello.c*. The file extension *.c* indicates that the file contains C language source code.

NOTE *If you've purchased an ebook, cut and paste the program into the editor. Use cut and paste when possible, as it can cut down on transcription errors.*

Compiling and Running Your Program

Next we need to compile and run the program, which involves two separate steps. You can choose from numerous C compilers, and the command to compile the program depends on which compiler you're using. On Linux and other Unix-like operating systems, you can invoke the system compiler with the `cc` command. To compile your program, enter **`cc`** on the command line followed by the name of the file you want to compile:

```
% cc hello.c
```

NOTE *These commands are specific to Linux (and other Unix-like operating systems). Other compilers on other operating systems will need to be invoked differently. Refer to the documentation for your specific compiler.*

If you entered the program correctly, the compile command will create a new file called *a.out* in the same directory as your source code. Inspect your directory by using the `ls` command and you should see the following:

```
% ls
a.out  hello.c
```

The *a.out* file is the executable program, which you can now run on the command line:

```
% ./a.out
Hello, world!
```

If everything goes right, the program should print `Hello, world!` to the terminal window. If it doesn't, compare the program text from Listing 1-1 to your program to ensure they are the same.

The `cc` command has numerous flags and compiler options. The `-o file` flag, for example, lets you give the executable file a memorable name instead of *a.out*. The following compiler invocation names the executable *hello*:

```
% cc -o hello hello.c
% ./hello
Hello, world!
```

Now we'll inspect the *hello.c* program line by line.

Preprocessor Directives

The first two lines of the *hello.c* program use the `#include` preprocessor directive, which behaves as if you replaced it with the contents of the specified file at the exact same location. We include the `<stdio.h>` and `<stdlib.h>` headers to access the functions declared in those headers, which we can then call from the program. The `puts` function is declared in `<stdio.h>`, and the `EXIT_SUCCESS` macro is defined in `<stdlib.h>`. As the filenames suggest, `<stdio.h>` contains the declarations for C Standard I/O functions, and `<stdlib.h>` contains the declarations for general utility functions. You need to include the declarations for any library functions that you use in your program.

The main Function

The main portion of the program, shown previously in Listing 1-1, starts with ❶:

```
int main(void) {
```

This line defines the `main` function that's called at program startup. The `main` function defines the main entry point for the program that's executed in a hosted environment when the program is invoked from the command line or from another program. C defines two possible execution environments: *freestanding* and *hosted*. A freestanding environment may not provide an operating system and is typically used in embedded programming. These implementations provide a minimal set of library functions, and the name and type of the function called at program startup are implementation defined. This book primarily assumes a hosted environment.

We define `main` to return a value of type `int` and place `void` inside the parenthesis to indicate that the function does not accept arguments. The `int` type is a signed integer type that can be used to represent both positive and negative integer values as well as zero. Similar to other procedural languages, C programs consist of procedures (referred to as *functions*) that can accept arguments and return values. Each function is a reusable unit of work that you can invoke as frequently as necessary in your program. In this case, the

value returned by the `main` function indicates whether the program terminated successfully. The actual work performed by this particular function ❷ is to print out the line `Hello, world!`:

```
  puts("Hello, world!");
```

The `puts` function is a C Standard Library function that writes a string argument to `stdout`, which typically represents the console or terminal window, and appends a newline character to the output. `"Hello, world!"` is a string literal that behaves like a read-only string. This function invocation outputs `Hello, world!` to the terminal.

Once your program has completed, you'll want it to exit. You can exit the program using the `return` statement ❸ within `main` to return an integer value to the host environment or invoking script:

```
  return EXIT_SUCCESS;
```

`EXIT_SUCCESS` is an object-like macro that commonly expands to 0 and is typically defined as follows:

```
#define EXIT_SUCCESS 0
```

Each occurrence of `EXIT_SUCCESS` is replaced by a 0, which is then returned to the host environment from the call to `main`. The script that invokes the program can then check its status to determine whether the invocation was successful. A return from the initial call to the `main` function is equivalent to calling the C Standard Library `exit` function with the value returned by the `main` function as its argument.

The final line of this program ❹ consists of a closing brace }, which closes the code block we started with the declaration of the `main` function:

```
int main(void) {
  // ---snip---
}
```

You can place the opening brace on the same line as the declaration or its own line, as follows:

```
int main(void)
{
  // ---snip---
}
```

This decision is strictly a stylistic one. In this book, I generally place the opening brace on the line with the function declaration because it's stylistically more compact.

Checking Function Return Values

Functions will often return a value that's the result of a computation or that signifies whether the function successfully completed its task. For example, the `puts` function we used in our "Hello, world!" program takes a string to print and returns a value of type `int`. The `puts` function returns the value of the macro `EOF` (a negative integer) if a write error occurs; otherwise, it returns a nonnegative integer value.

Although it's unlikely that the `puts` function will fail and return `EOF` for our simple program, it's possible. Because the call to `puts` can fail and return `EOF`, it means that your first C program has a bug or, at least, can be improved as follows:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  if (puts("Hello, world!") == EOF) {
    return EXIT_FAILURE;
    // code here never executes
  }
  return EXIT_SUCCESS;
  // code here never executes
}
```

This revised version of the "Hello, world!" program checks whether the `puts` call returns the value `EOF`, indicating a write error. If the function returns `EOF`, the program returns the value of the `EXIT_FAILURE` macro (which evaluates to a nonzero value). Otherwise, the function succeeds, and the program returns `EXIT_SUCCESS` (which is required to be 0). The script that invokes the program can then check its status to determine whether it was successful. Code following a return statement is *dead code* that never executes.

Formatted Output

The `puts` function is a nice, simple way to write a string to `stdout`, but eventually you'll need to print formatted output by using the `printf` function—for example, to print arguments other than strings. The `printf` function takes a format string that defines how the output is formatted, followed by a variable number of arguments that are the actual values you want to print. For example, if you want to use the `printf` function to print out `Hello, world!`, you could write it like this:

```
printf("%s\n", "Hello, world!");
```

The first argument is the format string `"%s\n"`. The `%s` is a conversion specification that instructs the `printf` function to read the second argument (a string literal) and print it to `stdout`. The `\n` is an alphabetic escape sequence used to represent nongraphic characters, and tells the function to

include a new line after the string. Without the newline sequence, the next characters printed (likely the command prompt) would appear on the same line. This function call outputs the following:

```
Hello, world!
```

Take care not to pass user-supplied data as part of the first argument to the `printf` function, because doing so can result in a formatted output security vulnerability (Seacord 2013).

The simplest way to output a string is to use the `puts` function, as previously shown. If you do use `printf` instead of `puts` in the revised version of the "Hello, world!" program, you'll find it no longer works, because the `printf` function returns status differently than the `puts` function. The `printf` function returns the number of characters printed if it's successful, or a negative value if an output or encoding error occurred. You can try modifying the "Hello, world!" program to use the `printf` function as an exercise.

Editors and Integrated Development Environments

A variety of editors and IDEs can be used to develop your C programs. Figure 1-1 shows the most used editors, according to a 2018 JetBrains survey.

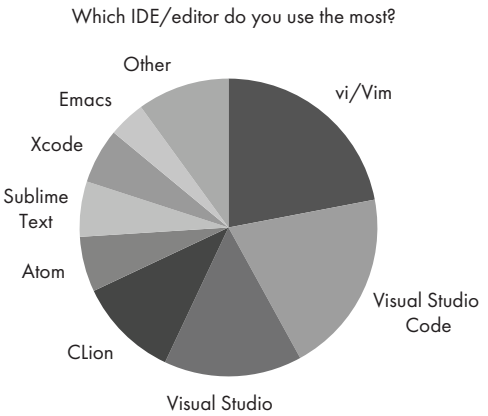

Figure 1-1: IDE/editor use

The exact tools available depend on which system you're using. This book focuses on Linux, Windows, and macOS, as these are the most common development platforms.

For Microsoft Windows, Microsoft's Visual Studio IDE (*https://visualstudio.microsoft.com/*) is an obvious choice. Visual Studio comes in three editions: Community, Professional, and Enterprise. The Community edition has the advantage of being free, while the other editions add features at a cost. For this book, you'll need only the Community edition.

For Linux, the choice is less obvious. Vim, Emacs, Visual Studio Code, and Eclipse are all options. Vim is the editor of choice for many developers and power users. It is a text editor based on the vi editor written by Bill Joy in the 1970s for a version of Unix. It inherits the key bindings of vi but also adds functionality and extensibility that are missing from the original vi. You can optionally install Vim plug-ins such as YouCompleteMe (*https://github.com/Valloric/YouCompleteMe/*) or deoplete (*https://github.com/Shougo/deoplete.nvim/*) that provide native semantic code completion for C programming.

GNU Emacs is an extensible, customizable, and free text editor. At its core, it's an interpreter for Emacs Lisp, a dialect of the Lisp programming language with extensions to support text editing—although I've never found this to be a problem. Full disclosure: almost all the C code I've ever developed was edited in Emacs.

Visual Studio Code (VS Code) is a streamlined code editor with support for development operations such as debugging, task running, and version control (covered in Chapter 11). It provides just the tools a developer needs for a quick code-build-debug cycle. VS Code runs on macOS, Linux, and Windows and is free for private or commercial use. Installation instructions are available for Linux and other platforms;[4] on Windows, you'd most likely use Microsoft's Visual Studio. Figure 1-2 shows Visual Studio Code being used to develop the "Hello, world!" program from Listing 1-1 on Ubuntu. You can see from the debug console that the program exited with status code 0 as expected.

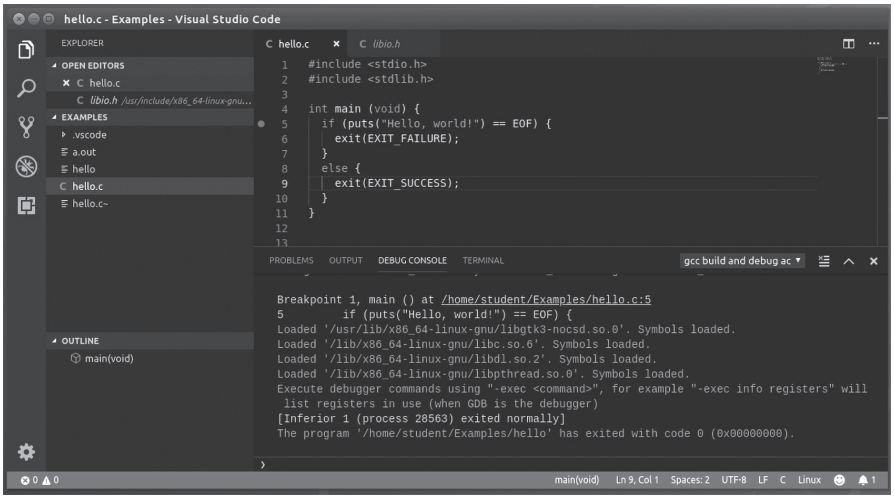

Figure 1-2: Visual Studio Code running on Ubuntu

4. See"Visual Studio Code on Linus" at *https://code.visualstudio.com/docs/setup/linux/* for installation instruction s on Linux.

Compilers

Many C compilers are available, so I won't discuss them all here. Different compilers implement different versions of the C Standard. Many compilers for embedded systems support only C89/C90. Popular compilers for Linux and Windows work harder to support modern versions of the C Standard, up to and including support for C2x.

GNU Compiler Collection

The *GNU Compiler Collection (GCC)* includes frontends for C, C++, and Objective-C, as well as other languages *(https://gcc.gnu.org/)*. GCC development follows a well-defined development plan under the guidance of the GCC steering committee.

GCC has been adopted as the standard compiler for Linux systems, although versions are also available for Microsoft Windows, macOS, and other platforms. Installing GCC on Linux is easy. The following command, for example, should install GCC 8 on Ubuntu:

```
% sudo apt-get install gcc-8
```

You can test the version of GCC you're using with the following command:

```
% gcc --version
gcc (Ubuntu 8.3.0-6ubuntu1~18.04) 8.3.0
This is free software; see the source for copying conditions. There is NO
Warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Fedora is an ideal development system if you intend to develop software for Red Hat Enterprise Linux. The following command can be used to install GCC on Fedora:

```
% sudo dnf install gcc
```

Clang

Another popular compiler is *Clang* (*https://clang.llvm.org/*). Installing Clang on Linux is also easy. The following command, for example, should install Clang on Ubuntu:

```
% sudo apt-get install clang
```

You can test the version of Clang you're using with the following command:

```
% clang --version
clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

Microsoft Visual Studio

The most popular development environment for Windows is *Microsoft Visual Studio*, which includes both the IDE and the compiler. The most recent version of Visual Studio at the time of writing is 2019 (*https://visualstudio.microsoft.com/downloads/*). It's bundled with Visual C++ 2019, which includes both the C and C++ compilers.

You can set options for Visual Studio on the Project Property Pages. On the Advanced tab under C/C++, make sure you compile as C code by using the Compile as C Code (`/TC`) option and not the Compile as C++ Code (`/TP`) option. By default, when you name a file with a *.c* extension, it's compiled with `/TC`. If the file is named with a *.cpp*, *.cxx*, or a few other extensions, it's compiled with `/TP`.

Portability

Every C compiler implementation is at least a little different. Compilers continually evolve—so, for example, a compiler like GCC might provide full support for C17 but be working toward support for C2x, in which case it might have some C2x features implemented but not others. Consequently, compilers support a full spectrum of C Standard versions (including in-between versions). The overall evolution of C implementations is slow, with many compilers significantly lagging behind the C Standard.

Programs written for C can be considered *strictly conforming* if they use only those features of the language and library specified in the standard. These programs are intended to be maximally portable. However, because of the range of implementation behaviors, no real-world C program is strictly conforming, nor ever will be (and probably shouldn't be). Instead, the C Standard allows you to write *conforming* programs that may depend on nonportable language and library features.

It's common practice to write code for a single reference implementation, or sometimes several implementations, depending on which platforms you plan to deploy your code. The C Standard is what ensures that these implementations don't differ too much, and allows you to target several at once, without having to learn a new language each time.

Five kinds of portability issues are enumerated in Annex J of the C Standard documents:

- Implementation-defined behavior
- Unspecified behavior
- Undefined behavior
- Locale-specific behavior
- Common extensions

As you learn about the C language, you'll encounter examples of all five kinds of behaviors, so it's important to understand precisely what these are.

Implementation-Defined Behavior

Implementation-defined behavior is program behavior that's not specified by the C Standard and that may offer different results among implementations, but has consistent, documented behavior within an implementation. An example of implementation-defined behavior is the number of bits in a byte.

Implementation-defined behaviors are mostly harmless but can cause defects when porting to different implementations. Whenever possible, avoid writing code that depends on implementation-defined behaviors that vary among the C implementations you might use to compile your code. A complete list of implementation-defined behaviors is enumerated in Annex J.3 of the C Standard. You can document your dependencies on these implementation-defined behaviors by using a `static_assert` declaration, as discussed in Chapter 11. Throughout this book, I note when code has implementation-defined behavior.

Unspecified Behavior

Unspecified behavior is program behavior for which the standard provides two or more options. The standard imposes no requirements on which option is chosen in any instance. Each execution of a given expression may have different results or produce a different value than a previous execution of the same expression. An example of unspecified behavior is function parameter storage layout, which can vary among function invocations within the same program. Avoid writing code that depends on the unspecified behaviors enumerated in Annex J.1 of the C Standard.

Undefined Behavior

Undefined behavior is behavior that isn't defined by the C Standard, or less circularly, "behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the standard imposes no requirements." Examples of undefined behavior include signed integer overflow and dereferencing an invalid pointer value. Code that has undefined behavior is often erroneous, but is more nuanced than that. Undefined behaviors are identified in the standard as follows:

- When a "shall" or "shall not" requirement is violated, and that requirement appears outside a constraint, the behavior is undefined
- When behavior is explicitly specified by the words "undefined behavior"
- By the omission of any explicit definition of behavior

The first two kinds of undefined behavior are frequently referred to as *explicit undefined behaviors*, while the third kind is referred to as *implicit*

undefined behavior. There is no difference in emphasis among these three; they all describe behavior that is undefined. The C Standard Annex J.2, "Undefined behavior," contains a list of explicit undefined behaviors in C.

Developers frequently misunderstand undefined behaviors to be errors or omissions in the C Standard, but the decision to classify a behavior as undefined is *intentional* and *considered*. Behaviors are classified as undefined by the C Standards committee to do the following:

- Give the implementer license not to catch program errors that are difficult to diagnose
- Avoid defining obscure corner cases that would favor one implementation strategy over another
- Identify areas of possible conforming language extension in which the implementer may augment the language by providing a definition of the officially undefined behavior

These three reasons are really quite different but are all considered portability issues. We'll examine examples of all three as they come up over the course of this book. Compilers (implementations) have the latitude to do the following:

- Ignore undefined behavior completely, giving unpredictable results
- Behave in a documented manner characteristic of the environment (with or without issuing a diagnostic)
- Terminate a translation or execution (with issuing a diagnostic)

None of these options are great (particularly the first), so it's best to avoid undefined behaviors except when the implementation specifies these behaviors are defined to allow you to invoke a language augmentation.[5]

Locale-Specific Behavior and Common Extensions

Locale-specific behavior depends on local conventions of nationality, culture, and language that each implementation documents. Common extensions are widely used in many systems but are not portable to all implementations.

Summary

In this chapter, you learned how to write a simple C language program, compile it, and run it. We then looked at several editors and interactive development environments as well as a few compilers that you can use to develop C programs on Windows, Linux, and macOS systems. Generally speaking, you should use newer versions of the compilers and other tools,

5. Compilers sometimes have a `pedantic` mode that can help notify the programmer of these portability issues.

as they tend to support newer features of the C programming language and provide better diagnostics and optimizations. You may not want to use newer versions of compilers if they break your existing code or if you're getting ready to deploy your code, to avoid introducing unnecessary changes into your already tested application. We concluded this chapter with a discussion of the portability of C language programs.

Subsequent chapters will examine specific features of the C language and library, starting with objects, functions, and types in the next chapter.

2

OBJECTS, FUNCTIONS, AND TYPES

In this chapter, you'll learn about objects, functions, and types. We'll examine how to declare variables (objects with identifiers) and functions, take the addresses of objects, and dereference those object pointers. You already learned about object types that are available to C programmers, as well as derived types. The first thing you'll learn in this chapter is one of the last things that I learned: every type in C is either an *object* type or a *function* type.

Objects, Functions, Types, and Pointers

An *object* is storage in which you can represent values. To be precise, an object is defined by the C Standard (ISO/IEC 9899:2018) as a "region of

data storage in the execution environment, the contents of which can represent values," with the added note, "when referenced, an object can be interpreted as having a particular type." A variable is an example of an object.

Variables have a declared *type* that tells you the kind of object its value represents. For example, an object with type `int` contains an integer value. The type is important because the collection of bits that represents one type of object will likely have a different value if interpreted as a different type of object. For example, the number 1 is represented in IEEE 754 (the IEEE Standard for Floating-Point Arithmetic) by the bit pattern `0x3f800000` (IEEE 754–2008). But if you were to interpret this same bit pattern as an integer, you'd get the value 1,065,353,216 instead of 1.

Functions are not objects but do have types. A function type is characterized by both its return type as well as the number and types of its parameters.

The C language also has *pointers*, which can be thought of as an *address*—a location in memory where an object or function is stored. A pointer type is derived from a function or object type called the *referenced type*. A pointer type derived from the referenced type `T` is called a *pointer to* `T`.

Because objects and functions are different things, object pointers and function pointers are also different things, and should not be used interchangeably. In the following section, you'll write a simple program that attempts to swap the values of two variables to help you better understand objects, functions, pointers, and types.

Declaring Variables

When you declare a variable, you assign it a type and provide it a name, or *identifier*, by which to reference the variable.

Listing 2-1 declares two integer objects with initial values. This simple program also declares, but doesn't define, a `swap` function to swap those values.

```
#include <stdio.h>

❶ void swap(int, int); // defined in Listing 2-2

int main(void) {
  int a = 21;
  int b = 17;

❷ swap(a, b);
  printf("main: a = %d, b = %d\n", a, b);
  return 0;
}
```

Listing 2-1: Program meant to swap two integers

This example program shows a `main` function with a single code block between the { } characters. This kind of code block is also known as a *compound statement*. We define two variables, `a` and `b`, within the `main` function. We declare the variables as having the type `int` and initialize them to `21` and `17`, respectively. Each variable must have a declaration. The `main` function then calls the `swap` function ❷ to try to swap the values of the two integers. The `swap` function is declared in this program ❶ but not defined. We'll look at some possible implementations of this function later in this section.

DECLARING MULTIPLE VARIABLES

You can declare multiple variables in any single declaration, but doing so can get confusing if the variables are pointers or arrays, or the variables are different types. For example, the following declarations are all correct:

```
char *src, c;
int x, y[5];
int m[12], n[15][3], o[21];
```

The first line declares two variables, `src` and `c`, which have different types. The `src` variable has a type of `char *`, and `c` has a type of `char`. The second line again declares two variables, `x` and `y`, with different types. The variable `x` has a type `int`, and `y` is an array of five elements of type `int`. The third line declares three arrays—`m`, `n`, and `o`—with different dimensions and numbers of elements.

These declarations are easier to understand if each is on its own line:

```
char *src;    // src has a type of char *
char c;       // c has a type of char
int x;        // x has a type int
int y[5];     // y is an array of 5 elements of type int
int m[12];    // m is an array of 12 elements of type int
int n[15][3]; // n is an array of 15 arrays of 3 elements of type int
int o[21];    // o is an array of 21 elements of type int
```

Readable and understandable code is less likely to have defects.

Swapping Values (First Attempt)

Each object has a storage duration that determines its *lifetime*, which is the time during program execution for which the object exists, has storage, has a constant address, and retains its last-stored value. Objects must not be referenced outside their lifetime.

Local variables such as `a` and `b` from Listing 2-1 have *automatic storage duration*, meaning that they exist until execution leaves the block in which they're defined. We are going to attempt to swap the values stored in these two variables.

Listing 2-2 is our first attempt to implement the `swap` function.

```
void swap(int a, int b) {
  int t = a;
  a = b;
  b = t;
  printf("swap: a = %d, b = %d\n", a, b);
}
```

Listing 2-2: The swap function

The `swap` function declares two parameters, `a` and `b`, that you use to pass arguments to this function. C distinguishes between *parameters*, which are objects declared as part of the function declaration that acquire a value on entry to the function, and *arguments*, which are comma-separated expressions you include in the function call expression. We also declare a temporary variable `t` of type `int` in the `swap` function and initialize it to the value of `a`. This variable is used to temporarily save the value stored in `a` so that it is not lost during the swap.

You can now compile and test the complete program by running the generated executable:

```
% ./a.out
swap: a = 17, b = 21
main: a = 21, b = 17
```

This result may be surprising. The variables `a` and `b` were initialized to `21` and `17`, respectively. The first call to `printf` within the `swap` function shows these two values swapped, but the second call to `printf` in `main` shows the original values unchanged. Let's examine what happened.

C is a *call-by-value* (also called a *pass-by-value*) language, which means that when you provide an argument to a function, the value of that argument is copied into a distinct variable for use within the function. The `swap` function assigns the values of the objects you pass as arguments to the respective parameters. When the values of the parameters in the function are changed, the values in the caller are unaffected because they are distinct objects. Consequently, the variables `a` and `b` retain their original values in `main` during the second call to `printf`. The goal of the program was to swap the values of these two objects. By testing the program, we've discovered it has a bug, or defect.

Swapping Values (Second Attempt)

To repair this bug, you can use pointers to rewrite the `swap` function. We use the indirection (*) operator to both declare pointers and dereference them, as shown in Listing 2-3.

```
void swap(int *pa, int *pb) {
  int t = *pa;
  *pa = *pb;
```

```
  *pb = t;
  return;
}
```

Listing 2-3: The revised swap function using pointers

When used in a function declaration or definition, * acts as part of a pointer declarator indicating that the parameter is a pointer to an object or function of a specific type. In the rewritten `swap` function, we specify two parameters, `pa` and `pb`, and declare them both as type pointers to `int`.

When you use the unary * operator in expressions within the function, the unary * operator dereferences the pointer to the object. For example, consider the following assignment:

```
pa = pb;
```

This replaces the value of the pointer `pa` with the value of the pointer `pb`. Now consider the actual assignment in the `swap` function:

```
*pa = *pb;
```

This dereferences the pointer `pb`, reads the referenced value, dereferences the pointer `pa`, and then overwrites the value at the location referenced by `pa` with the value referenced by `pb`.

When you call the `swap` function in `main`, you must also place an ampersand (`&`) character before each variable name:

```
swap(&a, &b);
```

The unary `&` is the *address-of* operator, which generates a pointer to its operand. This change is necessary because the `swap` function now accepts pointers to objects of type `int` as parameters instead of simply values of type `int`.

Listing 2-4 shows the entire `swap` program with emphasis on the objects created during execution of this code and their values.

```
#include <stdio.h>
void swap(int *pa, int *pb) {   // pa → a: 21    pb → b: 17
  int t = *pa;                  // t: 21
  *pa = *pb;                    // pa → a: 17    pb → b: 17
  *pb = t;                      // pa → a: 17    pb → b: 21

}
int main(void) {
  int a = 21;                   // a: 21
  int b = 17;                   // b: 17
  swap(&a, &b);
  printf("a = %d, b = %d\n", a, b);    // a: 17    b: 21
  return 0;
}
```

Listing 2-4: Simulated call-by-reference

Upon entering the `main` block, the variables `a` and `b` are initialized to `21` and `17`, respectively. The code then takes the addresses of these objects and passes them to the `swap` function as arguments.

Within the `swap` function, the parameters `pa` and `pb` now both declared to have the type pointer to `int` and contain copies of the arguments passed to `swap` from the calling function (in this case, `main`). These address copies still refer to the exact same objects, so when the values of the objects they reference are swapped in the `swap` function, the contents of the original objects declared in `main` are accessed and also swapped. This approach simulates *call-by-reference* (also known as *pass-by-reference*) by generating object addresses, passing those by value, and then dereferencing the copied addresses to access the original objects.

Scope

Objects, functions, macros, and other C language identifiers have *scope* that delimits the contiguous region where they can be accessed. C has four types of scope: file, block, function prototype, and function.

The scope of an object or function identifier is determined by where it is declared. If the declaration is outside any block or parameter list, the identifier has *file scope*, meaning the scope is the entire text file in which it appears as well as any files included after that point.

If the declaration appears inside a block or within the list of parameters, it has *block scope*, meaning that the identifier it declares is accessible only within the block. The identifiers for `a` and `b` from Listing 2-4 have block scope and can be used to refer to only these variables within the code block in the `main` function in which they're defined.

If the declaration appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator. *Function scope* is the area between the opening { of a function definition and its closing }. A label name is the only kind of identifier that has function scope. *Labels* are identifiers followed by a colon and identify a statement in a function to which control may be transferred. Chapter 5 covers labels and control transfer.

Scopes can be *nested,* with *inner* and *outer* scopes. For example, you can have a block scope inside another block scope, and every block scope is defined within a file scope. The inner scope has access to the outer scope, but not vice versa. As the name implies, any inner scope must be completely contained within the outer scopes that encompass it.

If you declare the same identifier in both the inner scope and an outer scope, the identifier declared in the outer scope is *hidden* by the identifier within the inner scope, which takes precedence. In this case, naming the identifier will refer to the object in the inner scope; the object from the outer scope is hidden and cannot be referenced by its name. The easiest way to prevent this from becoming a problem is to use different names.

Listing 2-5 demonstrates different scopes and how identifiers declared in inner scopes can hide identifiers declared in outer scopes.

```
int j;  // file scope of j begins

void f(int i) {         // block scope of i begins
  int j = 1;            // block scope of j begins; hides file-scope j
  i++;                  // i refers to the function parameter
  for (int i = 0; i < 2; i++) {  // block scope of loop-local i begins
    int j = 2;          // block scope of the inner j begin; hides outer j
    printf("%d\n", j);  // inner j is in scope, prints 2
  }                     // block scope of the inner i and j ends
  printf("%d\n", j);    // the outer j is in scope, prints 1
}  // the block scope of i and j ends

void g(int j);          // j has function prototype scope; hides file-scope j
```

Listing 2-5: Scope

There is nothing wrong with this code, provided the comments accurately describe your intent. Best practice is to use different names for different identifiers to avoid confusion, which leads to bugs. Using short names such as `i` and `j` is fine for identifiers with small scopes. Identifiers in large scopes should have longer, descriptive names that are unlikely to be hidden in nested scopes. Some compilers will warn about hidden identifiers.

Storage Duration

Objects have a storage duration that determines their lifetime. Altogether, four storage durations are available: automatic, static, thread, and allocated. You've already seen that objects of automatic storage duration are declared within a block or as a function parameter. The lifetime of these objects begins when the block in which they're declared begins execution, and ends when execution of the block ends. If the block is entered recursively, a new object is created each time, each with its own storage.

NOTE *Scope and lifetime are entirely different concepts.* Scope *applies to identifiers, whereas* lifetime *applies to objects. The scope of an identifier is the code region where the object denoted by the identifier can be accessed by its name. The lifetime of an object is the time period for which the object exists.*

Objects declared in file scope have *static* storage duration. The lifetime of these objects is the entire execution of the program, and their stored value is initialized prior to program startup. You can also declare a variable within a block scope to have static storage duration by using the storage-class specifier `static`, as shown in the counting example in Listing 2-6. These objects persist after the function has exited.

```
void increment(void) {
  static unsigned int counter = 0;
  counter++;
  printf("%d ", counter);
}

int main(void) {
  for (int i = 0; i < 5; i++) {
    increment();
  }
  return 0;
}
```

Listing 2-6: A counting example

This program outputs `1 2 3 4 5`. We initialize the static variable `counter` to 0 once at program startup, and increment it each time the `increment` function is called. The lifetime of `counter` is the entire execution of the program, and it will retain its last-stored value throughout its lifetime. You could achieve the same behavior by declaring `counter` with file scope. However, it is good software engineering practice to limit the scope of an object wherever possible.

Static objects must be initialized with a constant value and not a variable:

```
int *func(int i) {
  const int j = i; // ok
  static int k = j; // error
  return &k;
}
```

A constant value refers to literal constants (for example, `1`, `'a'`, or `0xFF`), `enum` members, and the results of operators such as `alignof` or `sizeof`; not `const`-qualified objects.

Thread storage duration is used in concurrent programming and is not covered by this book. *Allocated* storage duration deals with dynamically allocated memory and is discussed in Chapter 6.

Alignment

Object types have alignment requirements that place restrictions on the addresses at which objects of that type may be allocated. An *alignment* represents the number of bytes between successive addresses at which a given object can be allocated. CPUs may have different behavior when accessing aligned data (for example, the data address is a multiple of the data size) versus unaligned data.

Some machine instructions can perform multibyte accesses on non-word boundaries, but there may be a performance penalty. Some platforms cannot access unaligned memory. Alignment requirements may depend on the CPU word size (typically, 16, 32, or 64 bits).

Generally, C programmers need not concern themselves with alignment requirements, because the compiler chooses suitable alignments for its various types. Dynamically allocated memory from `malloc` is required to be sufficiently aligned for all standard types, including arrays and structures. However, on rare occasions, you might need to override the compiler's default choices; for example, to align data on the boundaries of the memory cache lines that must start at power-of-two address boundaries, or to meet other system-specific requirements. Traditionally, these requirements were met by linker commands, or by overallocating memory with `malloc` followed by rounding the user address upward, or similar operations involving other nonstandard facilities.

C11 introduced a simple, forward-compatible mechanism for specifying alignments. Alignments are represented as values of the type `size_t`. Every valid alignment value is a nonnegative integral power of two. An object type imposes a default alignment requirement on every object of that type: a stricter alignment (a larger power of two) can be requested using the alignment specifier (`_Alignas`). You can include an alignment specifier in the declaration specifiers of a declaration. Listing 2-7 uses the alignment specifier to ensure that `good_buff` is properly aligned (`bad_buff` may have incorrect alignment for member-access expressions).

```
struct S {
  int i; double d; char c;
};

int main(void) {
  unsigned char bad_buff[sizeof(struct S)];
  _Alignas(struct S) unsigned char good_buff[sizeof(struct S)];

  struct S *bad_s_ptr = (struct S *)bad_buff;   // wrong pointer alignment
  struct S *good_s_ptr = (struct S *)good_buff; // correct pointer alignment
}
```

Listing 2-7: Use of the `_Alignas` keyword

Alignments are ordered from weaker to stronger (also called stricter) alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any valid, weaker alignment requirement.

Object Types

This section introduces the object types in C. Specifically, we'll cover the Boolean type, character types, and numerical types (including both integer and floating-point types).

Boolean Types

Objects declared as `_Bool` can store only the values 0 and 1. This *Boolean type* was introduced in C99, and starts with an underscore to differentiate it in

existing programs that had already declared their own identifiers named `bool` or `boolean`. Identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved. The idea is that the C Standards committee can create new keywords such as `_Bool`, assuming that you have avoided the use of reserved identifiers. If you haven't, as far as the C Standards committee is concerned, it is your fault for not reading the standard carefully.

If you include the header `<stdbool.h>`, you can also spell this type as `bool` and assign it the values `true` (which expands to the integer constant `1`) and `false` (which expands to the integer constant `0`). Here we declare two Boolean variables using both spellings of the type name:

```
#include <stdbool.h>
_Bool flag1 = 0;
bool flag2 = false;
```

Both spellings will work, but it is better to use `bool`, as this is the long-term direction for the language.

Character Types

The C language defines three *character types*: `char`, `signed char`, and `unsigned char`. Each compiler implementation will define `char` to have the same alignment, size, range, representation, and behavior as either `signed char` or `unsigned char`. Regardless of the choice made, `char` is a separate type from the other two and is incompatible with both.

The `char` type is commonly used to represent character data in C language programs. In particular, objects of type `char` must be able to represent the minimum set of characters required in the execution environment (known as the *basic execution character set*), including upper- and lowercase letters, the 10 decimal digits, the space character, and various punctuation and control characters. The `char` type is inappropriate for integer data; it is safer to use `signed char` to represent small signed integer values, and `unsigned char` to represent small unsigned values.

The basic execution character set suits the needs of many conventional data processing applications, but its lack of non-English letters is an obstacle to acceptance by international users. To address this need, the C Standards committee specified a new, wide type to allow large character sets. You can represent the characters of a large character set as *wide characters* by using the `wchar_t` type, which generally takes more space than a basic character. Typically, implementations choose 16 or 32 bits to represent a wide character. The C Standard Library provides functions that support both narrow and wide character types.

Numerical Types

C provides several *numerical types* that can be used to represent integers, enumerators, and floating-point values. Chapter 3 covers some of these in more detail, but here's a brief introduction.

Integer Types

Signed integer types can be used to represent negative numbers, positive numbers, and zero. The signed integer types include `signed char`, `short int`, `int`, `long int`, and `long long int`.

Except for `int` itself, the keyword `int` may be omitted in the declarations for these types, so you might, for example, declare a type by using `long long` instead of `long long int`.

For each signed integer type, there is a corresponding *unsigned integer type* that uses the same amount of storage: `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int`. The unsigned types can be used to represent only positive numbers and zero.

The signed and unsigned integer types are used to represent integers of various sizes. Each platform (current or historical) determines the size for each of these types, given some constraints. Each type has a minimum representable range. The types are ordered by width, guaranteeing that *wider* types are at least as large as *narrower* types so that an object of type `long long int` can represent all values that an object of type `long int` can represent, an object of type `long int` can represent all values that can be represented by an object of type `int`, and so forth.

The `int` type usually has the natural size suggested by the architecture of the execution environment, so the size would be 16 bits wide on a 16-bit architecture, and 32 bits wide on a 32-bit architecture. You can specify actual-width integers by using type definitions from the `<stdint.h>` or `<inttypes.h>` headers, like `uint32_t`. These headers also provide type definitions for the widest available integer types: `uintmax_t` and `intmax_t`.

Chapter 3 covers integer types in excruciating detail.

enum Types

An *enumeration*, or `enum`, allows you to define a type that assigns names (*enumerators*) to integer values in cases with an enumerable set of constant values. The following are examples of enumerations:

```
enum day { sun, mon, tue, wed, thu, fri, sat };
enum cardinal_points { north = 0, east = 90, south = 180, west = 270 };
enum months { jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
```

If you don't specify a value to the first enumerator with the `=` operator, the value of its enumeration constant is 0, and each subsequent enumerator without an `=` adds 1 to the value of the previous enumeration constant. Consequently, the value of `sun` in the `day` enumeration is 0, `mon` is 1, and so forth.

You can also assign specific values to each enumerator, as shown by the `cardinal_points` enumeration. Using `=` with enumerators may produce enumeration constants with duplicate values, which can be a problem if you incorrectly assume all the values are unique. The `months` enumeration sets the first enumerator at 1, and each subsequent enumerator that isn't specifically assigned a value will be incremented by 1.

The actual value of the enumeration constant must be representable as an `int`, but its type is implementation defined. For example, Visual C++ uses a `signed int`, and GCC uses an `unsigned int`.

Floating-Point Types

The C language supports three *floating-point types*: `float`, `double`, and `long double`. Floating-point arithmetic is similar to, and often used as a model for, the arithmetic of real numbers. The C language supports a variety of floating-point representations including, on most systems, the IEEE Standard for Floating-Point Arithmetic (IEEE 754–2008). The choice of floating-point representation is implementation dependent. Chapter 3 covers floating-point types in detail.

void Types

The `void` type is a rather strange type. The keyword `void` (by itself) means "cannot hold any value." For example, you can use it to indicate that a function doesn't return a value, or as the sole parameter of a function to indicate that the function takes no arguments. On the other hand, the *derived type* `void *` means that the pointer can reference *any* object. I'll discuss derived types later in this chapter.

Function Types

Function types are derived types. In this case, the type is derived from the return type and the number and types of its parameters. The return type of a function cannot be an array type.

When you declare a function, you use the *function declarator* to specify the name of the function and the return type. If the declarator includes a parameter type list and a definition, the declaration of each parameter must include an identifier, except parameter lists with only a single parameter of type `void`, which needs no identifier.

Here are a few function type declarations:

```
int f(void);
int *fip();
void g(int i, int j);
void h(int, int);
```

First, we declare a function `f` with no parameters that returns an `int`. Next, we declare a function `fip` with no specified parameters that returns a pointer to an `int`. Finally, we declare two functions, `g` and `h`, each returning `void` and taking two parameters of type `int`.

Specifying parameters with identifiers (as done here with `g`) can be problematic if an identifier is a macro. However, providing parameter

names is good practice for self-documenting code, so omitting the identifiers (as done with `h`) is not typically recommended.

In a function declaration, specifying parameters is optional. However, failing to do so is occasionally problematic. If you were to write the function declaration for `fip` in C++, it would declare a function accepting no arguments and returning an `int *`. In C, `fip` declares a function accepting any number of arguments of any type and returning an `int *`. You should never declare functions with an empty parameter list in C. First, this is a deprecated feature of the language that may be removed in the future. Second, the code could be ported to C++, so explicitly list parameter types and use `void` when there are no parameters.

A function type with a parameter type list is known as a *function prototype*. A function prototype informs the compiler about the number and types of parameters a function accepts. Compilers use this information to verify that the correct number and type of parameters are used in the *function definition* and any calls to the function.

The function definition provides the actual implementation of the function. Take a look at the following function definition:

```
int max(int a, int b)
{ return a > b ? a : b; }
```

The return type specifier is `int`; the function declarator is `max(int a, int b)`; and the function body is `{ return a > b ? a : b; }`. The specification of a function type must not include any type qualifiers (see "Type Qualifiers" on page 32). The function body itself uses the condition operator (`? :`), which is explained further in Chapter 4. This expression states that if `a` is greater than `b`, return `a`; otherwise, return `b`.

Derived Types

Derived types are types that are constructed from other types. These include pointers, arrays, type definitions, structures, and unions, all of which we'll cover here.

Pointer Types

A *pointer type* is derived from the function or object type that it points to, called the *referenced* type. A pointer provides a reference to an entity of the referenced type.

The following three declarations declare a pointer to `int`, a pointer to `char`, and a pointer to `void`:

```
int *ip;
char *cp;
void *vp;
```

Earlier in the chapter, I introduced the address-of (`&`) and indirection (*) operators. You use the `&` operator to take the address of an object or function. If the object is an `int`, for example, the result of the operator has the type pointer to `int`:

```
int i = 17;
int *ip = &i;
```

We declare the variable `ip` as a pointer to `int` and assign it the address of `i`. You can also use the `&` operator on the result of the * operator:

```
ip = &*ip;
```

Dereferencing `ip` by using the indirection operator resolves to the actual object `i`. Taking the address of `*ip` by using the `&` operator retrieves the pointer, so these two operations cancel each other out.

The unary * operator converts a pointer to a type into a value of that type. It denotes *indirection* and operates only on pointers. If the operand points to a function, the result of using the * operator is the function designator, and if it points to an object, the result is a value of the designated object. For example, if the operand is a pointer to `int`, the result of the indirection operator has the type `int`. If the pointer is not pointing to a valid object or function, bad things may happen.

Arrays

An *array* is a contiguously allocated sequence of objects that all have the same element type. Array types are characterized by their element types and the number of elements in the array. Here we declare an array of 11 elements of type `int` identified by `ia`, and an array of 17 elements of type pointer to `float` identified by `afp`:

```
int ia[11];
float *afp[17];
```

You use square brackets ([]) to identify an element of an array. For example, the following contrived code snippet creates the string `"0123456789"` to demonstrate how to assign values to the elements of an array:

```
char str[11];
for (unsigned int i = 0; i < 10; ++i) {
❶ str[i] = '0' + i;
}
str[10] = '\0';
```

The first line declares an array of `char` with a bound of 11. This allocates sufficient storage to create a string with 10 characters plus a null character.

The `for` loop iterates 10 times, with the values of `i` ranging from 0 to 9. Each iteration assigns the result of the expression `'0' + i` to `str[i]`. Following the end of the loop, the null character is copied to the final element of the array `str[10]`.

In the expression at ❶, `str` is automatically converted to a pointer to the first member of the array (an object of type `char`), and `i` has an unsigned integer type. The subscript (`[]`) operator and addition (`+`) operator are defined so that `str[i]` is identical to `*(str + i)`. When `str` is an array object (as it is here), the expression `str[i]` designates the `i`th element of the array (counting from 0). Because arrays are indexed starting at 0, the array `char str[11]` is indexed from 0 to 10, with 10 being the last element, as referenced on the last line of this example.

If the operand of the unary `&` operator is the result of a `[]` operator, the result is as if the `&` operator were removed and the `[]` operator were changed to a `+` operator. For example, `&str[10]` is the same as `str + 10`.

You can also declare multidimensional arrays. Listing 2-8 declares `arr` in the function `main` as a two-dimensional 5 × 3 array of type `int`, also referred to as a *matrix*.

```
void func(int arr[5]);
int main(void) {
  unsigned int i = 0;
  unsigned int j = 0;
  int arr[3][5];
❶ func(arr[i]);
❷ int x = arr[i][j];
  return 0;
}
```

Listing 2-8: Matrix operations

More precisely, `arr` is an array of three elements, each of which is an array of five elements of type `int`. When you use the expression `arr[i]` at ❶ (which is equivalent to `*(arr+i)`), the following occurs:

1. `arr` is converted to a pointer to the initial array of five elements of type `int` starting at `arr[i]`.
2. `i` is scaled to the type of `arr` by multiplying `i` by the size of one array of five `int` objects.
3. The results from steps 1 and 2 are added.
4. Indirection is applied to the result to produce an array of five elements of type `int`.

When used in the expression `arr[i][j]` at ❷, that array is converted to a pointer to the first element of type `int`, so `arr[i][j]` produces an object of type `int`.

TYPE DEFINITIONS

You use a typedef to declare an alias for an existing type; it never creates a new type. For example, each of the following declarations creates a new type alias:

```
typedef unsigned int uint_type;
typedef signed char schar_type, *schar_p, (*fp)(void);
```

On the first line, we declare uint_type as an alias for the type unsigned int. On the second line, we declare schar_type as an alias for signed char, schar_p as an alias for signed char *, and fp as an alias for signed char(*)(void). Identifiers that end in _t in the standard headers are type definitions (aliases for existing types). Generally speaking, you should not follow this convention in your own code because the C Standard reserve identifiers that match the patterns int[0-9a-z_]*_t and uint[0-9a-z_]*_t, and the Portable Operating System Interface (POSIX) reserves all identifiers that end in _t. If you define identifiers that use these names, they may collide with names used by the implementation, which can cause problems that are difficult to debug.

Structures

A *structure type* (also known as a struct) contains sequentially allocated member objects. Each object has its own name and may have a distinct type—unlike arrays, which must all be of the same type. Structures are similar to record types found in other programming languages. Listing 2-9 declares an object identified by sigline that has a type of struct sigrecord and a pointer to the sigline object identified by sigline_p.

```
struct sigrecord {
  int signum;
  char signame[20];
  char sigdesc[100];
} sigline, *sigline_p;
```

Listing 2-9: struct sigrecord

The structure has three member objects: signum is an object of type int, signame is an array of type char consisting of 20 elements, and sigdesc is an array of type char consisting of 100 elements.

Structures are useful for declaring collections of related objects and may be used to represent things such as a date, customer, or personnel record. They are especially useful for grouping objects that are frequently passed together as arguments to a function, so you don't need to repeatedly pass individual objects separately.

Once you have defined a structure, you'll likely want to reference its members. You reference members of an object of the structure type by using the structure member (`.`) operator. If you have a pointer to a structure, you can reference its members with the structure pointer (`->`) operator. Listing 2-10 demonstrates the use of each operator.

```
sigline.signum = 5;
strcpy(sigline.signame, "SIGINT");
strcpy(sigline.sigdesc, "Interrupt from keyboard");

❶ sigline_p = &sigline;

sigline_p->signum = 5;
strcpy(sigline_p->signame, "SIGINT");
strcpy(sigline_p->sigdesc, "Interrupt from keyboard");
```

Listing 2-10: Referencing structure members

The first three lines of Listing 2-10 directly access members of the `sigline` object by using the `.` operator. At ❶, we assign the pointer to `sigline_p` to the address of the `sigline` object. In the final three lines of the program, we indirectly access the members of the `sigline` object by using the `->` operator through the `sigline_p` pointer.

Unions

Union types are similar to structures, except that the memory used by the member objects overlaps. Unions can contain an object of one type at one time, and an object of a different type at a different time, but never both objects at the same time, and are primarily used to save memory. Listing 2-11 shows the union `u` that contains three structures: `n`, `ni`, and `nf`. This union might be used in a tree, graph, or other data structure that has some nodes that contain integer values (`ni`) and other nodes that contain floating-point values (`nf`).

```
union {
  struct {
    int type;
  } n;
  struct {
    int type;
    int intnode;
  } ni;
  struct {
    int type;
    double doublenode;
  } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
```

Listing 2-11: Unions

As with structures, you can access union members via the `.` operator. Using a pointer to a union, you can reference its members with the `->` operator. In Listing 2-11, the `type` member in the `nf` struct of the union is referenced as `u.nf.type`, and the `doublenode` member is referenced as `u.nf.doublenode`. Code that uses this union will typically check the type of the node by examining the value stored in `u.n.type` and then accessing either the `intnode` or `doublenode` `struct` depending on the type. If this had been implemented as a structure, each node would contain storage for both the `intnode` and the `doublenode` members. The use of a union allows the same storage to be used for both members.

Tags

Tags are a special naming mechanism for structs, unions, and enumerations. For example, the identifier `s` appearing in the following structure is a tag:

```
struct s {
  //---snip---
};
```

By itself, a tag is not a type name and cannot be used to declare a variable (Saks 2002). Instead, you must declare variables of this type as follows:

```
struct s v;   // instance of struct s
struct s *p;  // pointer to struct s
```

The names of unions and enumerations are also tags and not types, meaning that they cannot be used alone to declare a variable. For example:

```
enum day { sun, mon, tue, wed, thu, fri, sat };
day today;  // error
enum day tomorrow;  // OK
```

The tags of structures, unions, and enumerations are defined in a separate *namespace* from ordinary identifiers. This allows a C program to have both a tag and another identifier with the same spelling in the same scope:

```
enum status { ok, fail };  // enumeration
enum status status(void);  // function
```

You can even declare an object `s` of type `struct s`:

```
struct s s;
```

This may not be good practice, but it is valid in C. You can think of `struct` tags as type names and define an alias for the tag by using a `typedef`. Here's an example:

```
typedef struct s { int x; } t;
```

This now allows you to declare variables of type `t` instead of `struct s`. The tag name in `struct`, `union`, and `enum` is optional, so you can just dispense with it entirely:

```
typedef struct { int x; } t;
```

This works fine except in the case of self-referential structures that contain pointers to themselves:

```
struct tnode {
  int count;
  struct tnode *left;
  struct tnode *right;
};
```

If you omit the tag on the first line, the compiler may complain because the referenced structure on lines 3 and 4 has not yet been declared, or because the whole structure is not used anywhere. Consequently, you have no choice but to declare a tag for the structure, but you can declare a `typedef` as well:

```
typedef struct tnode {
  int count;
  struct tnode *left;
  struct tnode *right;
} tnode;
```

Most C programmers use a different name for the tag and the `typedef`, but the same name works just fine. You can also define this type before the structure so that you can use it to declare the `left` and `right` members that refer to other objects of type `tnode`:

```
typedef struct tnode tnode;
struct tnode {
  int count;
  tnode *left
  tnode *right;
} tnode;
```

Type definitions can improve code readability beyond their use with structures. For example, all three of the following declarations of the `signal` function specify the same type:

```
typedef void fv(int), (*pfv)(int);
void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

Type Qualifiers

All the types examined so far have been unqualified types. Types can be *qualified* by using one or more of the following qualifiers: `const`, `volatile`, and `restrict`. Each of these qualifiers changes behaviors when accessing objects of the qualified type.

The qualified and unqualified versions of types can be used interchangeably as arguments to functions, return values from functions, and members of unions.

NOTE *The `_Atomic` type qualifier, available since C11, supports concurrent programs.*

const

Objects declared with the `const` qualifier (`const`-qualified types) are not modifiable. In particular, they're not assignable but can have constant initializers. This means objects with `const`-qualified types can be placed in read-only memory by the compiler, and any attempt to write to them will result in a runtime error:

```
const int i = 1; // const-qualified int
i = 2; // error: i is const-qualified
```

It's possible to accidentally convince your compiler to change a `const`-qualified object for you. In the following example, we take the address of a `const`-qualified object `i` and tell the compiler that this is actually a pointer to an `int`:

```
const int i = 1;  // object of const-qualified type
int *ip = (int *)&i;
*ip = 2;  // undefined behavior
```

C does not allow you to cast away the `const` if the original was declared as a `const`-qualified object. This code might appear to work, but it's defective and may fail later. For example, the compiler might place the `const`-qualified object in read-only memory, causing a memory fault when trying to store a value in the object at runtime.

C allows you to modify an object that is pointed to by a `const`-qualified pointer by casting the `const` away, provided that the original object was not declared `const`:

```
int i = 12;
const int j = 12;
const int *ip = &i;
const int *jp = &j;
*(int *)ip = 42; // ok
*(int *)jp = 42; // undefined behavior
```

volatile

Objects of volatile-qualified types serve a special purpose. Static volatile-qualified objects are used to model memory-mapped input/output (I/O) ports, and static constant volatile-qualified objects model memory-mapped input ports such as a real-time clock.

The values stored in these objects may change without the knowledge of the compiler. For example, every time the value from a real-time clock is read, it may change, even if the value has not been written to by the C program. Using a volatile-qualified type lets the compiler know that the value may change, and ensures that every access to the real-time clock occurs (otherwise, an access to the real-time clock may be optimized away or replaced by a previously read and cached value). In the following code, for example, the compiler must generate instructions to read the value from port and then write this value back to port:

```
volatile int port;
port = port;
```

Without the volatile qualification, the compiler would see this as a *no-op* (a programming statement that does nothing) and potentially eliminate both the read and the write.

Also, volatile-qualified types are used for communications with signal handlers and with setjmp/longjmp (refer to the C Standard for information on signal handlers and setjmp/longjmp). Unlike in Java and other programming languages, volatile-qualified types in C should not be used for synchronization between threads.

restrict

A restrict-qualified pointer is used to promote optimization. Objects indirectly accessed through a pointer frequently cannot be fully optimized because of potential *aliasing*, which occurs when more than one pointer refers to the same object. Aliasing can inhibit optimizations, because the compiler can't tell if portions of an object can change values when another apparently unrelated object is modified, for example.

The following function copies n bytes from the storage referenced by q to the storage referenced by p. The function parameters p and q are both restrict-qualified pointers:

```
void f(unsigned int n, int * restrict p, int * restrict q) {
  while (n-- > 0) {
    *p++ = *q++;
  }
}
```

Because both p and q are restrict-qualified pointers, the compiler can assume that an object accessed through one of the pointer parameters is not also accessed through the other. The compiler can make this

assessment based solely on the parameter declarations without analyzing the function body. Although using `restrict`-qualified pointers can result in more efficient code, you must ensure that the pointers do not refer to overlapping memory to prevent undefined behavior.

Exercises

Try these code exercises on your own:

1. Add a `retrieve` function to the counting example from Listing 2-6 to retrieve the current value of `counter`.
2. Declare an array of three pointers to functions and invoke the appropriate function based on an index value passed in as an argument.

Summary

In this chapter, you learned about objects and functions and how they differ. You learned how to declare variables and functions, take the addresses of objects, and dereference those object pointers. You also learned about most of the object types that are available to C programmers as well as derived types.

We'll return to these types in later chapters to explore in more detail how they can be best used to implement your designs. In the next chapter, I provide detailed information about the two kinds of arithmetic types: integers and floating-point.

3

ARITHMETIC TYPES

In this chapter, you'll learn about the two kinds of *arithmetic types*: integers and floating-point types. Most operators in C operate on arithmetic types. Because C is a system-level language, performing arithmetic operations correctly can be difficult—resulting in frequent defects. This is partially because arithmetic operations in digital systems with limited range and precision do not always produce the same result as they would in ordinary mathematics. Being able to perform basic arithmetic correctly in C is an essential foundation to becoming a professional C programmer.

We'll dive deep into how arithmetic works in the C language so that you have a firm grasp of these fundamental concepts. We'll also look at how to convert one arithmetic type to another, which is necessary for performing operations on mixed types.

Integers

As mentioned in Chapter 2, each integer type represents a finite range of integers. Signed integer types represent values that can be negative, zero, or positive; unsigned integers represent values that can be only zero or positive. The range that each type of integer can represent depends on your implementation.

The *value* of an integer object is the ordinary mathematical value stored in the object. The *representation* of a value for an integer object is the particular encoding of the value in the bits of the object's allocated storage. We'll look at the representation in more detail later.

Padding and Precision

All integer types except `char`, `signed char`, and `unsigned char` may contain unused bits, called *padding*, that allow implementations to accommodate hardware quirks (such as skipping over a sign bit in the middle of a multiple-word representation) or to optimally align with a target architecture. The number of bits used to represent a value of a given type, excluding padding but including the sign, is called the *width* and is often denoted by *N*. The *precision* is the number of bits used to represent values, excluding sign and padding bits.

The <limits.h> Header File

The `<limits.h>` header file provides the minimum and maximum representable values for the various integer types. A *representable value* is one that can be represented in the number of bits available to an object of a particular type. Values that cannot be represented will be diagnosed by the compiler or converted to a representable but different (incorrect) value. Compiler writers provide the correct minimum, maximum, and width values for their implementations. To write portable code, you should use these constants, rather than integer literals such as +2147483647 that represent a specific limit and may change when porting to a different implementation.

The C Standard imposes only three constraints on integer sizes. First, storage for *every* data type occupies an integral number of adjacent `unsigned char` objects (which may include padding). Second, each integer type has to support the minimum ranges, allowing you to depend on a portable range of values across any implementation. Third, smaller types cannot be wider than larger types. So, for example, `USHRT_MAX` cannot be greater than `UINT_MAX`, but they can be the same width.

Declaring Integers

Unless explicitly declared as `unsigned`, integer types are assumed to be signed (except for `char`, which the implementation can define as either a signed or unsigned integer type). The following are valid declarations of unsigned integers:

```
unsigned int ui; // unsigned is required
unsigned u; // int can be omitted
unsigned long long ull2; // int can be omitted
unsigned char uc; // unsigned is required
```

When declaring signed integer types, you can omit the `signed` keyword—except for `signed char`, which requires the keyword to distinguish `signed char` from plain `char`. Unless it is the only keyword present, `int` can also be omitted. For example, instead of declaring a variable to be of type `signed long long int`, it is common practice to just declare it as `long long` and save some typing. The following are all valid declarations of signed integers:

```
int i; // signed can be omitted
long long int sll; // signed can be omitted
long long sll2; // signed and int can be omitted
signed char sc; // signed is required
```

Unsigned Integers

Unsigned integers have ranges that start at 0, and their upper bound is greater than that of the corresponding signed integer type. Unsigned integers are frequently used for counting items that may have large, nonnegative quantities.

Representation

Unsigned integer types are easier to understand and to use than signed integer types. They represent values using a pure binary system with no offset: the least significant bit has the weight 2^0, the next least significant has the weight 2^1, and so forth. The value of the binary number is the sum of all the weights for the set bits. Table 3-1 shows some examples of unsigned values using an unpadded 8-bit representation.

Table 3-1: 8-Bit Unsigned Values

Decimal	Binary	Hexadecimal
0	0000 0000	0x00
1	0000 0001	0x01
17	0001 0001	0x11
255	1111 1111	0xFF

Unsigned integer types do not require the sign to be represented and so generally provide 1 bit greater precision than the corresponding signed integer types. Unsigned integer values range from 0 to a maximum value that depends on the width of the type. This maximum value is $2^N - 1$, where N is the width. For example, most x86 architectures use 32-bit integers with no padding bits, so an object of type `unsigned int` has a range of 0 to $2^{32} - 1$ (4,294,967,295). The constant expression `UINT_MAX` from `<limits.h>` specifies the implementation-defined upper range for this type. Table 3-2 shows the constant expressions from `<limits.h>` for each unsigned type, the minimum range required by the standard, and the actual range on modern x86 implementations.

Table 3-2: Unsigned Integer Ranges

Constant expression	Minimum magnitudes	x86	Maximum value for an object of type
`UCHAR_MAX`	255 // $2^8 - 1$	Same	`unsigned char`
`USHRT_MAX`	65,535 // $2^{16} - 1$	Same	`unsigned short int`
`UINT_MAX`	65,535 // $2^{16} - 1$	4,294,967,295	`unsigned int`
`ULONG_MAX`	4,294,967,295 // $2^{32} - 1$	Same	`unsigned long int`
`ULLONG_MAX`	18,446,744,073,709,551,615 // $2^{64} - 1$	Same	`unsigned long long int`

Wraparound

Wraparound occurs when you perform arithmetic operations that result in values too small (less than 0) or too large (greater than $2^N - 1$) to be represented as a particular unsigned integer type. In this case, the value is reduced modulo the number that is one greater than the largest value that can be represented in the resulting type. Wraparound is well-defined behavior in the C language. Whether it is a defect in your code depends on the context. If you are counting something and the value wraps, it is likely to be an error. However, the use of wraparound in certain encryption algorithms is intentional.

For example, the code in Listing 3-1 initializes `ui` to its maximum value and then increments it. The resulting value cannot be represented as an `unsigned int`, so it wraps around to 0. If this value is then decremented, it falls outside the range once more, so it wraps around again to `UINT_MAX`.

```
unsigned int ui = UINT_MAX;  // 4,294,967,295 on x86
ui++;
printf("ui = %u\n", ui); // ui is 0
ui--;
printf("ui = %u\n", ui); // ui is 4,294,967,295
```

Listing 3-1: Unsigned integer wraparound

Because of wraparound, an unsigned integer expression can never evaluate to less than 0. It's easy to lose track of this and implement comparisons that are always true or always false. For example, the `i` in the following `for` loop can never take on a negative value, so this loop will never terminate:

```
for (unsigned int i = n; i >= 0; --i)
```

This behavior has caused some notable real-world bugs. For example, all six power-generating systems on a Boeing 787 are managed by a corresponding generator control unit. Boeing's laboratory testing discovered that an internal software counter in the generator control unit wraps around after running continuously for 248 days, according to the Federal Aviation Administration.[6] This defect causes all six generator control units on the engine-mounted generators to enter fail-safe mode at the same time.

To avoid unplanned behavior (such as having your airplane fall from the sky), it's important to check for wraparound by using the limits from `<limits.h>`. You should be careful when implementing these checks, because it is easy to make mistakes. For example, the following code contains a defect as `sum + ui` can never be larger than `UINT_MAX`:

```
extern unsigned int ui, sum;
// assign values to ui and sum
if (sum + ui > UINT_MAX)
  too_big();
else
  sum = sum + ui;
```

If the result of `sum + ui` is larger than `UINT_MAX`, it's reduced modulo `UINT_MAX + 1`. Therefore, this entire test is useless, and the generated code will unconditionally perform the summation. Quality compilers might issue a warning pointing this out, but not all do. To remedy this, we can subtract `sum` from both sides of the inequality to form the following effective test:

```
extern unsigned int ui, sum;
// assign values to ui and sum
if (ui > UINT_MAX - sum)
  too_big();
else
  sum = sum + ui;
```

`UINT_MAX` is the largest value that can be represented as an `unsigned int`, and `sum` is a value between 0 and `UINT_MAX`. If `sum` is equal to `UINT_MAX`, the result of the subtraction is 0, and if `sum` is equal to 0, the result of the subtraction is `UINT_MAX`. Because the result of this operation will always fall in the allowable range of 0 to `UINT_MAX`, it can never wrap.

6. See *Airworthiness Directives; The Boeing Company Airplanes, https://www.federalregister.gov/d/2015-10066/.*

The same problem occurs when checking the result of an arithmetic operation against 0, the minimum unsigned value:

```
extern unsigned int i, j;
// assign values to i and j
if (i - j < 0)  // cannot happen
  negative();
else
  i = i - j;
```

Because unsigned integer values can never be negative, the subtraction will be performed unconditionally. Quality compilers may warn about this mistake as well. Instead of this useless test, we can check for wraparound by testing whether `j` is greater than `i`:

```
if (j > i)  // correct
  negative();
else
  i = i - j;
```

If `j > i`, the result would wrap around, so the possibility of wraparound is clearly detected. By eliminating the subtraction operation in the test, we eliminate the possibility of wraparound occurring during the test.

WARNING *Keep in mind that the width used when wrapping depends on the implementation, which means you can obtain different results on different platforms. Unless you take this into account, your code won't be portable.*

Signed Integers

Each unsigned integer type (excluding `_Bool`) has a corresponding signed integer type that occupies the same amount of storage. We use signed integers to represent negative, zero, and positive values, the range of which depends on the number of bits allocated to the type and the representation.

Representation

Representing signed integer types is more complicated than representing unsigned integer types. Historically, the C language has supported three signed integer representation schemes:

Sign and magnitude The high-order bit indicates the sign, and the remaining bits represent the magnitude of the value in pure binary notation.

Ones' complement The sign bit is given the weight $-(2^{N-1} - 1)$, and the other value bits have the same weights as for unsigned.

Two's complement The sign bit is given the weight $-(2^{N-1})$, and the other value bits have the same weights as for unsigned.

You cannot choose which representation to use; that is determined by the implementers of C for the various systems. Though all three are still in use, two's complement is by far the most common representation—so much so that the C Standards committee intends, starting with C2x, to accommodate only two's-complement representation. The remainder of this book assumes two's-complement representation.

Signed integer types with a width of N can represent any integer value in the range of -2^{N-1} to $2^{N-1}-1$. This means, for example, that an 8-bit value of type `signed char` has a range of –128 to 127. Two's complement can represent an additional *most negative* value. The most negative value for an 8-bit `signed char` is –128, and its absolute value |–128| cannot be represented as this type. This leads to some interesting edge cases, which we'll examine later in this chapter and the next.

Table 3-3 shows the constant expressions from `<limits.h>` for each signed type, the minimum range required by the standard, and the actual range on modern x86 implementations.

Table 3-3: Signed Integer Ranges

Constant expression	Minimum magnitudes	x86	Type
`SCHAR_MIN`	–127 // $-(2^7-1)$	–128	`signed char`
`SCHAR_MAX`	+127 // 2^7-1	Same	`signed char`
`SHRT_MIN`	–32,767 // $-(2^{15}-1)$	–32,768	`short int`
`SHRT_MAX`	+32,767 // $2^{15}-1$	Same	`short int`
`INT_MIN`	–32,767 // $-(2^{15}-1)$	–2,147,483,648	`int`
`INT_MAX`	+32,767 // $2^{15}-1$	+2,147,483,647	`int`
`LONG_MIN`	–2,147,483,647 // $-(2^{31}-1)$	–2,147,483,648	`long int`
`LONG_MAX`	+2,147,483,647 // $2^{31}-1$	Same	`long int`
`LLONG_MIN`	–9,223,372,036,854,775,807 // $-(2^{63}-1)$	-9,223,372,036,854,775,808	`long long int`
`LLONG_MAX`	+9,223,372,036,854,775,807 // $2^{63}-1$	Same	`long long int`

The two's complement representation for negative signed values includes a sign bit and other value bits. The sign bit is given the weight $-(2^{N-1})$. To negate a value in two's-complement representation, simply toggle each non-padding bit and then add 1 (with carries as necessary), as shown in Figure 3-1.

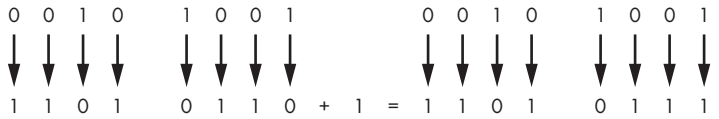

Figure 3-1: Negating an 8-bit value in two's-complement representation

Table 3-4 shows the binary and decimal representations for an 8-bit two's-complement signed integer type with no padding (that is, $N = 8$). This is not required knowledge, but as a C programmer, you will likely find it useful.

Table 3-4: 8-Bit Two's-Complement Values

Binary	Decimal	Weighting	Constant
00000000	0	0	
00000001	1	2^0	
01111110	126	$2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$	
01111111	127	$2^{N-1} - 1$	SCHAR_MAX
10000000	–128	$-(2^{N-1}) + 0$	SCHAR_MIN
10000001	–127	$-(2^{N-1}) + 1$	
11111110	–2	$-(2^{N-1}) + 126$	
11111111	–1	$-(2^{N-1}) + 127$	

Overflow

Overflow occurs when a signed integer operation results in a value that cannot be represented in the resulting type. For example, the following implementation of a function-like macro that returns the absolute value of an integer can overflow:

```
// undefined or wrong for most negative value
#define Abs(i) ((i) < 0 ? -(i) : (i))
```

We'll examine macros in detail in Chapter 9. For now, you can think of function-like macros as functions that operate on generic types. On the surface, this macro appears to correctly implement the absolute value function by returning the nonnegative value of `i` without regard to its sign. We use the conditional (`?:`) operator (which I'll cover in more detail in the next chapter) to test whether the value of `i` is negative. If so, `i` is negated to `-(i)`; otherwise, it evaluates to the unmodified value `(i)`.

Because we've implemented `Abs` as a function-like macro, it can take an argument of any type. Of course, invoking this macro with an unsigned integer is pointless, because unsigned integers can never be negative, so the macro's output would just reproduce the argument. However, we can invoke the function with a variety of signed integer and floating-point types, as in the following invocation:

```
signed int si = -25;
signed int abs_si = Abs(si);
printf("%d\n", abs_si);  // prints 25
```

In this example, we pass an object of type `signed int` with the value –25 as an argument to the `Abs` macro. This invocation expands to the following:

```
signed int si = -25;
signed int abs_si = ((si) < 0 ? -(si) : (si));
printf("%d\n", abs_si);  // prints 25
```

The macro correctly returned the absolute value of –25. So far, so good. The problem is that the negative of the two's-complement most negative value for a given type cannot be represented in that type, so this use of the `Abs` function results in signed integer overflow. Consequently, this implementation of `Abs` is defective and can do anything, including unexpectedly returning a negative value:

```
signed int si = INT_MIN;
signed int abs_si = Abs(si);  // undefined behavior
printf("%d\n", abs_si);
```

So, what should `Abs(INT_MIN)` return to fix this behavior? Signed integer overflow is undefined behavior in C, allowing implementations to silently wrap (the most common behavior), trap, or both. *Traps* interrupt execution of the program so that no further operations are performed. Common architectures like x86 do a combination of both. Because the behavior is undefined, no universally correct solution to this problem exists, but we can at least test for the possibility of undefined behavior before it occurs and take appropriate action.

To make the absolute-value macro useful for a variety of types, we'll add a type-dependent `flag` argument to it. The flag represents the `*_MIN` macro, which matches the type of the first argument. This value is returned in the problem case:

```
#define AbsM(i, flag) ((i) >= 0 ? (i) : ((i)==(flag) ? (flag) : -(i)))
signed int si = -25;  // try INT_MIN to trigger the problem case
signed int abs_si = AbsM(si, INT_MIN);
if (abs_si == INT_MIN)
  goto recover;  // special case
else
  printf("%d\n", abs_si);  // prints 25
```

The `AbsM` macro tests for the most negative value and simply returns it if found instead of triggering the undefined behavior by negating it.

On some systems, the C Standard library implements the following `int`-only absolute-value function to avoid overflow when the function is passed `INT_MIN` as an argument:

```
int abs(int i) {
  return (i >= 0) ? i : -(unsigned)i;  // avoids overflow
}
```

In this case, `i` is converted to an `unsigned int` and negated. I'll discuss conversions in more detail later in this chapter.

Perhaps surprisingly, unary minus (`-`) operator is defined for unsigned integer types. The resulting unsigned integer value is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type. Finally, `i` is implicitly converted back to `signed int` as required by the `return` statement. Because `-INT_MIN` can't be represented as a `signed int`, the result is implementation defined. This is why this implementation is used only on *some systems*, and even on these systems, the `abs` function returns an incorrect value.

The `Abs` and `AbsM` implementations use function-like macros to evaluate their parameters more than once. This can cause surprises when the arguments cause program state to change. These are called side effects and are covered in detail in the next chapter. Function calls, on the other hand, evaluate each argument only once.

Unsigned integers have well-defined wraparound behavior. Signed integer overflow, or the possibility of it, should always be considered a defect.

Integer Constants

Integer constants (or *integer literals*) are constants we use to introduce particular integer values into a program. For example, you might use them in a declaration or assignment to initialize a counter to 0. C has three kinds of integer constants that use different number systems: decimal constants, octal constants, and hexadecimal constants.

Decimal constants always begin with a nonzero digit. For example, the following code uses two decimal constants:

```
unsigned int ui = 71;
int si;
si = -12;
```

In this example code, we initialize `ui` to the decimal constant `71` and assign `si` the decimal constant value `-12`. Use decimal constants when introducing regular integer values into your code.

If a constant starts with a 0, optionally followed by digits 0 through 7, it is an *octal constant*. Here's an example:

```
int agent = 007;
int permissions = 0777;
```

In this example, `007` octal equals `7` decimal, and the octal constant `0777` equals the decimal value 511. Octal constants are convenient when dealing with 3-bit fields, for example.

You can also create a *hexadecimal constant* by prefixing a sequence of decimal digits and the letters a (or A) through f (or F) with `0x` or `0X`. For example:

```
int burger = 0xDEADBEEF;
```

Use hexadecimal constants when the constant you are introducing is meant to represent a bit pattern more than a particular value; for example, when representing an address. Idiomatically, most hexadecimal constants are written like `0xDEADBEEF` because it resembles a typical hex dump. It's probably a good idea for you to write all your hexadecimal constants like this.

You can also append a suffix to your constant to specify its type. Without a suffix, a decimal constant is given the `int` type if it can be represented as a value in that type. If it cannot be represented as an `int`, it will be represented as a `long int` or `long long int`. The suffixes are `U` for `unsigned`, `L` for `signed long`, and `LL` for `long long`. These can be combined. For example the `ULL` suffix represents the `unsigned long long` type. Here are some examples:

```
unsigned int ui = 71U;
signed long int sli = 9223372036854775807L;
unsigned long long int ui = 18446744073709551615ULL;
```

If we don't use a suffix, and the integer constant isn't of the required type, it may be implicitly converted. (We'll discuss implicit conversion in "Arithmetic Conversion" on page 49.) This may result in a surprising conversion or a compiler diagnostic, so it's best to correctly specify the desired type of your integer constants. Section 6.4.4.1 of the C Standard contains more information on integer constants.

Floating-Point

Floating-point is the most common representation for real numbers in computers. Floating-point representation is a technique that uses scientific notation to encode numbers with a base number and an exponent. For example, the decimal number `123.456` can be represented as 1.23456×10^2 while the binary number `0b10100.110` can be represented as 1.0100110×2^4.

You can generate floating-point representations in several ways. The C Standard doesn't require that implementations use any specific model, although it does require every implementation to support *some* model. To keep things simple, we'll assume conformance to Annex F (the most common floating-point format). You can test the values of the `__STDC_IEC_559__` or `__STDC_IEC_60559_BFP__` macros in newer compilers to determine whether the implementation conforms to Annex F.

This section explains floating-point types, arithmetic, values, and constants, so you will know how and when to use them to emulate math on real numbers and when to avoid them.

Floating-Point Types

C has three floating-point types: `float`, `double`, and `long double`.

The `float` type can be used for floating-point calculations in which the result can be adequately represented as a single-precision result. The common IEC 60559 `float` type encodes values using 1 sign bit, 8 exponent bits, and 23 significand bits (ISO/IEC/IEEE 60559:2011).

The `double` type provides greater precision but requires additional storage. It encodes values using 1 sign bit, 11 exponent bits, and 52 significand bits. These types are illustrated in Figure 3-2.

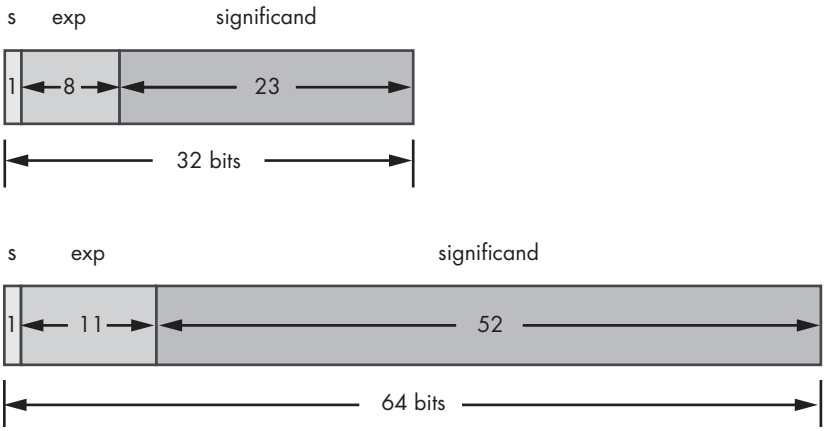

Figure 3-2: The `float` and `double` types

Each implementation assigns the `long double` type one of the following formats:

- IEC 60559 quadruple (or binary128) format[7]
- IEC 60559 binary64-extended format
- A non-IEC 60559 extended format
- IEC 60559 double (or binary64) format

Recommended practice for compiler implementers is to match the `long double` type with the IEC 60559 binary128 format or an IEC 60559 binary64-extended format. IEC 60559 binary64-extended formats include the common 80-bit IEC 60559 format.

Larger types have greater precision but require more storage. Any value that can be represented as a `float` can also be represented as a `double`, and any value that can be represented as a `double` can be represented as a `long double`. Future versions of the standard may include additional floating-point types, including those with greater range, precision, or both than `long double`, or lesser range and precision, such as 16-bit floating-point type.

7. IEC 60559 added binary128 to its basic formats in the 2011 revision.

SIGN, EXPONENT, AND SIGNIFICAND

As with integers, the *sign bit* represents whether the number is positive or negative: 0 denotes a positive number, and 1 denotes a negative number.

The *exponent field* needs to represent both positive and negative exponents. To avoid storing the exponent as a signed number, a bias is implicitly added to the actual exponent to get the *stored* exponent. For the `float` type, the bias is 127. Consequently, to express an exponent of 0, we would store 127 in the exponent field. A stored value of 200 indicates an exponent of 200 – 127, or 73. Exponents of –127 (where every exponent bit is 0) and +128 (where every exponent bit is 1) are reserved for special numbers. Similarly, the bias for double-precision numbers is 1023. This means that the value stored will range from 0 to 255 for a `float`, and 0 to 2047 for a `double`.

The *significand bits* represent the precision bits of the number. For example, if you were to represent the value 1.0100110×2^4 as a floating-point value, the significand refers to the precision bits 1.0100110, and the exponent refers to the power of 2, which is 4 in this example (Hollasch 2019).

Floating-Point Arithmetic

Floating-point arithmetic is similar to, and used to model, the arithmetic of real numbers. However, there are differences to consider. In particular, unlike the arithmetic of real numbers, floating-point numbers are bounded in magnitude and have finite precision. Addition and multiplication operations are *not* associative, the distributive property *doesn't* hold, nor do many other properties that are valid for real numbers.

Floating-point types cannot represent all real numbers exactly, even when they can be represented in a small number of decimal digits. For example, common decimal constants such as 0.1 can't be represented exactly as binary floating-point numbers. Floating-point types may lack the necessary precision for various applications such as loop counters or performing financial calculations. See CERT C rule FLP30-C (Do not use floating-point variables as loop counters) for more information.

Floating-Point Values

Ordinarily, all of the significand bits in a floating-point type express significant figures, in addition to a leading 1, which is implied and omitted, though still considered part of the value. As a special case, to represent the value 0, the exponent and significand must both be 0; zeros are signed (+0 and –0) according to the sign bit, so there are two floating-point zero values: a positive one and a negative one.

There are no leading zeros in the significand of a normal floating-point value; leading zeros are removed by adjusting the exponent. Therefore, `float` has 24 significant bits of precision, `double` has 53 significant bits of

precision, and `long double` has 113 significant bits of precision (assuming the quadruple 128-bit IEC 60559 format). These are *normalized* numbers, and they preserve the full precision of the significand.

Nonnormalized (or *subnormal*) numbers are very small positive and negative numbers (but not 0) whose representation would result in an exponent that is less than the smallest representable value. Figure 3-3 is a number line showing the range of subnormal values around 0. A nonzero number represented with the minimum exponent (that is, the implicit 1 bit is taken to be 0) is a subnormal, even if all its explicit significand bits are 1.

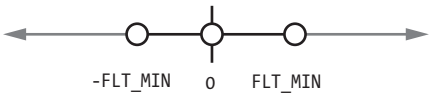

Figure 3-3: Domain of subnormal numbers

Floating-point types can also represent values that are not floating-point numbers, such as negative and positive infinity and not-a-number (NaN) values. NaNs are values that do not represent a real number.

Having infinity available as a specific value allows operations to continue past overflow situations, which often produces the desired result without requiring special treatment. For example, dividing any positive or negative nonzero value by either positive or negative zero[8] yields either positive or negative infinity. Operations with infinite values are well-defined in the IEEE floating-point standard.

A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception and is typically tested after a selected sequence of operations. A *signaling NaN* generally raises a floating-point exception immediately when occurring as an arithmetic operand. Floating-point exceptions are an advanced topic not covered here. For more information, refer to Annex F of the C Standard.

The `NAN` and `INFINITY` macros and the `nan` functions in `<math.h>` provide designations for IEC 60559 quiet NaNs and infinities. The `SNANF`, `SNAN`, and `SNANL` macros (ISO/IEC TS 18661-1:2014, ISO/IEC TS 18661-3:2015) defined in `<math.h>` provide designations for IEC 60559 signaling NaNs. The C Standard does not require full support for signaling NaNs.

One way to identify the kind of floating-point value you're dealing with is to use the `fpclassify` function-like macro, which classifies its argument value as NaN, infinite, normal, subnormal, or zero.

```
#include <math.h>
int fpclassify(real-floating x);
```

In Listing 3-2, we use the `fpclassify` macro in the `show_classification` function to determine whether a floating-point value of type `double` is a normal value, subnormal value, zero, infinity, or NaN.

8. Distinct values −0 and +0 compare as equal.

```
const char *show_classification(double x) {
  switch(fpclassify(x)) {
    case FP_INFINITE:  return "Inf";
    case FP_NAN:       return "NaN";
    case FP_NORMAL:    return "normal";
    case FP_SUBNORMAL: return "subnormal";
    case FP_ZERO:      return "zero";
    default:           return "unknown";
  }
```

Listing 3-2: The `fpclassify` macro

The function argument `x` (a `double` in this example) is passed to the `fpclassify` macro, which switches on the result. The function returns a string corresponding to the class of value stored in `x`.

Floating-Point Constants

A *floating-point constant* is a decimal or hexadecimal number that represents a signed real number. You should use floating-point constants to represent floating-point values that cannot be changed. The following are some examples of floating-point constants:

```
15.75
1.575E1   /* 15.75 */
1575e-2   /* 15.75 */
-2.5e-3   /* -0.0025 */
25E-4     /* 0.0025 */
```

All floating-point constants have a type. The type is `double` if unsuffixed, `float` if suffixed by the letter `f` or `F`, or `long double` if suffixed by the letter `l` or `L`, as shown here:

```
10.0   /* type double */
10.0F  /* type float */
10.0L  /* type long double */
```

Arithmetic Conversion

Frequently, a value represented in one type (for example, `float`) must be represented in a different type (for example, `int`). This might occur, for example, when you have an object of type `float` and need to pass it as an argument to a function that accepts only an object of type `int`. When such conversions are necessary, you should always ensure that the value is representable in the new type. I'll discuss this further in "Safe Conversions" on page 54.

Values can be implicitly or explicitly converted from one arithmetic type to another. You can use the *cast* operator to perform *explicit* conversions. Listing 3-3 shows two examples of casts.

```
int si = 5;
short ss = 8;
long sl = (long)si;❶
unsigned short us = (unsigned short)(ss + sl);❷
```

Listing 3-3: Cast operators

To perform a cast, place a type name in parentheses just before the expression. The cast converts the expression to the unqualified version of the type name in parentheses. Here, we cast the value of `si` to the type `long` ❶. Because `si` is of type `int`, this cast (from a smaller to a larger integer type of the same signedness) is guaranteed to be safe because the value can always be represented in the larger type.

The second cast in this code snippet ❷ casts the result of the expression (`ss + sl`) to type `unsigned short`. Because the value is converted to an unsigned type (`unsigned short`) with less precision, the result of the conversion might not be equal to the original value. (Some compilers might warn about this; others won't.) In this example, the result of the expression (13) can be correctly represented in the resulting type.

Implicit conversion, also known as *coercion*, occurs automatically in expressions as required. This happens, for example, when operations are performed on mixed types. In Listing 3-3, implicit conversions are used to convert `ss` to the type of `sl` so that the addition `ss + sl` can be performed on a common type. The rules concerning which values are implicitly converted to which types are somewhat complicated and involve three concepts: integer conversion rank, integer promotions, and the usual arithmetic conversions. We'll discuss these in the following sections.

Integer Conversion Rank

An *integer conversion rank* is a standard rank ordering of integer types used to determine a common type for computations. Every integer type has an integer conversion rank that determines when and how conversions are implicitly performed.

The C Standard, section 6.3.1.1, paragraph 1 (ISO/IEC 9899:2018), states that every integer type has an integer conversion rank where the following applies:

- No two signed integer types have the same rank, even if they have the same representation.
- The rank of a signed integer type is greater than the rank of any signed integer type with less precision.
- The rank of `long long int` is greater than the rank of `long int`, which is greater than the rank of `int`, which is greater than the rank of `short int`, which is greater than the rank of `signed char`.
- The rank of any unsigned integer type equals the rank of the corresponding signed integer type, if any.
- The rank of `char` equals the rank of `signed char` and `unsigned char`.

- The rank of `_Bool` is less than the rank of all other standard integer types.
- The rank of any enumerated type equals the rank of the compatible integer type. Each enumerated type is compatible with `char`, a signed integer type, or an unsigned integer type.
- The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined but still subject to the other rules for determining the integer conversion rank.

Integer Promotions

A *small type* is an integer with a lower conversion rank than `int` or `unsigned int`. *Integer promotion* is the process of converting values of small types to an `int` or `unsigned int`. Integer promotions allow you to use an expression of a small type in any expression where an `int` or `unsigned int` may be used. For example, you could use the lower-ranked integer type—typically, `char` or `short`—on the right-hand side of an assignment or as an argument to a function.

Integer promotions serve two primary purposes. First, they encourage operations to be performed in a natural size (`int`) for the architecture, which improves performance. Second, they help avoid arithmetic errors from the overflow of intermediate values, as shown in the following code snippet:

```
signed char cresult, c1, c2, c3;
c1 = 100; c2 = 3; c3 = 4;
cresult = c1 * c2 / c3;
```

Without integer promotion, `c1 * c2` would result in an overflow of the `signed char` type on platforms where `signed char` is represented by an 8-bit two's-complement value, because 300 is outside the range of values that can be represented in an object of this type, which is –128 to 127. However, because of integer promotion, `c1`, `c2`, and `c3` are implicitly converted to objects of type `signed int`, and the multiplication and division operations take place in this size. There is no possibility of overflow while performing these operations, because the resulting values can always be represented (objects of type `signed int` have a range of -2^{N-1} to $2^{N-1} - 1$). In this specific example, the result of the entire expression is 75, which is within range of the `signed char` type, so the value is preserved when stored in `cresult`.

Prior to the first C Standard, compilers used one of two approaches to integer promotions: the unsigned preserving approach or the value preserving approach. In the *unsigned preserving approach*, the compiler promotes unsigned small types to `unsigned int`. In the *value preserving approach*, if all values of the original type can be represented as an `int`, the value of the original small type will be converted to `int`. Otherwise, it is converted to `unsigned int`. When developing the original version of the standard (C89), the C Standards committee decided on value preserving rules, because they

produce incorrect results less often than the unsigned preserving approach. If necessary, you can override this behavior by using explicit type casts, as we did in Listing 3-3.

The result of promoting small unsigned types depends on the precision of the integer types, which is implementation-defined. For example, the x86 architecture has an 8-bit `char` type, a 16-bit `short` type, and a 32-bit `int` type. For implementations that target this architecture, values of both `unsigned char` and `unsigned short` are promoted to `signed int` because all the values that can be represented in these smaller types can be represented as a `signed int`. However, 16-bit architectures, such as Intel 8086/8088 and the IBM Series/1, have an 8-bit `char` type, a 16-bit `short` type, and a 16-bit `int` type. For implementations that target these architectures, values of type `unsigned char` are promoted to `signed int`, while values of type `unsigned short` are promoted to `unsigned int`. This is because all the values that can be represented as an 8-bit `unsigned char` type can be represented as a 16-bit `signed int`, but some values that can be represented as a 16-bit `unsigned short` cannot be represented as a 16-bit `signed int`.

Usual Arithmetic Conversions

The *usual arithmetic conversions* are rules for yielding a common type by balancing both operands of a binary operator to a common type, or balancing the second and third arguments of the conditional (`? :`) operator to a common type.

Balancing conversions changes one or both operands of different types to the same type. Many operators that accept integer operands—including `*`, `/`, `%`, `+`, `-`, `<`, `>`, `<=`, `>=`, `==`, `!=` , `&`, `^`, `|`, `? :`—perform conversions using the usual arithmetic conversions. The usual arithmetic conversions are applied to the promoted operands.

The usual arithmetic conversions first check whether one of the operands in the balancing conversion is a floating-point type. If so, it applies the following rules:

1. If one type of either operand is `long double`, the other operand is converted to `long double`.
2. Otherwise, if one type of either operand is `double`, the other operand is converted to `double`.
3. Otherwise, if the type of either operand is `float`, the other operand is converted to `float`.
4. Otherwise, the integer promotions are performed on both operands.

If one operand has the type `double` and the other operand has the type `int`, for example, the operand of type `int` is converted to an object of type `double`. If one operand has the type `float` and the other operand has the type `double`, the operand of type `float` is converted to an object of type `double`.

If neither operand is a floating-point type, the following usual arithmetic conversion rules are applied to the promoted integer operands:

1. If both operands have the same type, no further conversion is needed.
2. Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type that has the lesser integer conversion rank is converted to the type of the operand with greater rank. If one operand has the type `int` and the other operand has the type `long`, for example, the operand of type `int` is converted to an object of type `long`.
3. Otherwise, if the operand that has the unsigned integer type has a rank greater than or equal to the rank of the other operand's type, then the operand with the signed integer type is converted to the type of the operand with the unsigned integer type. For example, if one operand has the type `signed int`, and the other operand has the type `unsigned int`, the operand of type `signed int` is converted to an object of type `unsigned int`.
4. Otherwise, if the type of the operand with the signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type. For example, if one operand has the type `unsigned int` and the other operand has the type `signed long long`, and the `signed long long` type can represent all the values of the `unsigned int` type, then the operand of type `unsigned int` is converted to an object of type `signed long long`. This is the case for implementations with a 32-bit `int` type and a 64-bit `long long` type, such as x86-32 and x86-64.
5. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

These conversion rules, which evolved as new types were added in C's early years, take some getting used to. The irregularities in these patterns resulted from varying architectural properties (notably, the PDP-11's automatic promotion of `char` to `int`) coupled with a desire to avoid changing the behavior of existing programs, and (subject to those constraints) a desire for uniformity. When in doubt, use type casts to explicitly force the conversions that you intend. That said, try not to overuse explicit conversions because casts can disable important diagnostics.

An Example of Implicit Conversion

The following example illustrates the use of integer conversion rank, integer promotions, and the usual arithmetic conversions. This code compares the `signed char` value `c` for equality with the `unsigned int` value `ui`. We'll assume this code is being compiled for the x86 architecture:

```
unsigned int ui = UINT_MAX;
signed char c = -1;
if (c == ui) {
  puts("-1 equals 4,294,967,295");
}
```

The variable c is of type signed char. Because signed char has a lower integer conversion rank than int or unsigned int, the value stored in c is promoted to an object of type signed int when used in the comparison. This is accomplished by sign-extending the original value of 0xFF to 0xFFFFFFFF. *Sign extension* is used to convert a signed value to a larger-width object. The sign bit is copied into each bit position of the expanded object. This operation preserves the sign and magnitude when converting a value from a smaller to a larger, signed integer type.

Next, the usual arithmetic conversions are applied. Because the operands to the equal (==) operator have different signedness and equal rank, the operand with the signed integer type is converted to the type of the operand with the unsigned integer type. The comparison is then performed as a 32-bit unsigned operation. Because UINT_MAX has the same values as the promoted and converted value of c, the comparison yields 1, and the code snippet prints the following:

```
-1 equals 4,294,967,295
```

Safe Conversions

Both implicit and explicit conversions (the result of a cast operation) can produce values that can't be represented in the resulting type. It's preferable to perform operations on objects of the same type to avoid conversions. However, conversions are unavoidable when a function returns or accepts an object of a different type. In these cases, it is necessary to ensure that the conversion is performed correctly.

Integer Conversions

Integer conversions occur when a value of an integer type is converted to a different integer type. Conversions to larger types of the same signedness are always safe and don't need to be checked. Most other conversions can produce unexpected results if the resulting value cannot be represented in the resulting type. To perform these conversions correctly, you must test that the value stored in the original integer type is within the range of values that can be represented in the resulting integer type. As an example, the do_stuff function shown in Listing 3-4 accepts a signed long argument value that needs to be used in a context in which only a signed char is appropriate. To perform this conversion safely, the function checks that value can be represented as a signed char in the range [SCHAR_MIN, SCHAR_MAX] and returns an error if it is not.

```
#include <errno.h>
#include <limits.h>

errno_t do_stuff(signed long value) {
  if ((value < SCHAR_MIN) || (value > SCHAR_MAX)) {
```

```
    return ERANGE;
  }
  signed char sc = (signed char)value; // Cast quiets warning
  //---snip---
}
```

Listing 3-4: Safe conversion

The specific range tests vary based on the conversion. See CERT C rule INT31-C (Ensure that integer conversions do not result in lost or misinterpreted data) for more information.

Integer to Floating-Point Conversions

If the value of the integer type being converted to a floating-point type can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but not exactly, the result is rounded to either the nearest higher or nearest lower representable value, depending on the implementation. If the value being converted is outside the range of values that can be represented, the behavior is undefined. CERT C rule FLP36-C (Preserve precision when converting integral values to floating-point type) provides more information and examples of these conversions.

Floating-Point to Integer Conversions

When a finite value of a floating type is converted to an integer type (other than `bool`), the fractional part is discarded. If the value of the integral part cannot be represented by the integer type, the behavior is undefined.

Floating-Point Demotions

Converting a floating-point value to a larger floating-point type is always safe. Demoting a floating-point value (that is, converting to a smaller floating-point type) is similar to converting an integer value to a floating-point type.

Floating-point types that conform to Annex F support signed infinity. Demoting values of floating-point types for these implementations will always succeed because all values are within range. See CERT C rule FLP34-C (Ensure that floating-point conversions are within range of the new type) for more information on floating-point conversions.

Summary

In this chapter, you learned about integers and floating-point types. You also learned about implicit and explicit conversions, integer conversion rank, integer promotions, and the usual arithmetic conversions. In the next chapter, you'll learn about operators and how to write simple expressions to perform operations on various object types.

4

EXPRESSIONS AND OPERATORS

In this chapter, you'll learn about operators and how to write simple expressions to perform operations on various object types. An *operator* is a keyword or one or more punctuation characters used to perform an operation. When an operator is applied to one or more operands, it becomes an expression that computes a value and that might have side effects. *Expressions* are sequences of operators and operands that compute a value or accomplish another purpose. The operands can be identifiers, constants, string literals, and other expressions.

In this chapter, we discuss simple assignment before stepping back to describe the mechanics of expressions (operators and operands, value computations, side effects, precedence, and order of evaluation). We then discuss specific operators including arithmetic, bitwise, cast, conditional, alignment, relational, compound assignment, and the comma operator. We've introduced many of these operators and expressions in previous chapters; here, we detail their behavior and how best to use them. Finally, the chapter ends with a discussion of pointer arithmetic.

Simple Assignment

A *simple assignment* replaces the value stored in the object designated by the left operand with the right operand. The value of the right operand is converted to the type of the assignment expression. Simple assignment has three components: the left operand, the assignment (`=`) operator, and the right operand, as shown in the following example:

```
int i = 21; // declaration with initializer
int j = 7;  // declaration with initializer
i = j;      // simple assignment
```

The first two lines are *declarations* that define and initialize `i` with the value 21, and `j` with the value 7. An *initializer* uses an expression but is not itself an assignment expression, because an initializer is always part of a declaration.

The third line contains a simple assignment. You must define or declare all identifiers that appear in an expression such as simple assignment for your code to compile.

In simple assignment, the rvalue is converted to the type of the lvalue and then stored in the object designated by the lvalue. In the assignment `i = j`, the value is read from `j` and written to `i`. Because both `i` and `j` are the same type (`int`), no conversion is necessary. The assignment expression has the value of the result of the assignment and the type of the lvalue.

The left operand in simple assignment is always an expression (with an object type other than `void`), and we refer to it as an *lvalue*. The *l* in *lvalue* originally comes from it being the *left* operand, but it may be more correct to think of it as standing for *locator value*, because it must designate an object. In this example, the identifiers for both objects `i` and `j` are lvalues. An lvalue can also be an expression such as `*(p+4)`, provided it references an object in memory.

The right operand is also an expression but can simply be a value and does not need to identify an object. We refer to this value as an *rvalue* (*right* operand) or *expression value*. The rvalue does not need to refer to an object, as you can see in the following statement, which uses the types and values from the preceding example:

```
j = i + 12; // j now has the value 19
```

The expression `i + 12` is not an lvalue, because there is no underlying object storing the result. Instead, `i` by itself is an lvalue that is automatically converted into an rvalue to be used as an operand to the addition operation. The resulting value from the addition operation (which has no memory location associated with it) is also an rvalue. C constrains where lvalues and rvalues may appear. The following statements illustrate the correct and incorrect use of lvalues and rvalues:

```
int i;
i = 5;     // i is an lvalue, 5 is an rvalue
int j = i; // lvalues can appear on the right-hand side of an assignment
7 = i;     // error: rvalues can't appear on the left-hand side of an assignment
```

The assignment `7 = i` won't work, because the rvalue must always go on the right side of the operator. In the following example, the right operand has a different type than the assignment expression, so the value of `i` is first converted to a `signed char` type. The value of the expression enclosed in parentheses is then converted to the `long int` type:

```
signed char c;
int i = INT_MAX;
long k;
k = (c = i);
```

Assignment must deal with real-world constraints. Specifically, simple assignment can result in truncation if a value is converted to a narrower type. As mentioned in Chapter 3, each object requires a fixed number of bytes of storage. The value of `i` can always be represented by `k` (a larger type of the same signedness). However, in this example, the value of `i` is converted to `signed char` (the type of the assignment expression `c = i`). The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression—that is, `long int` type. Assuming `c` has fewer bits available to represent the value stored in `i`, values greater than `SCHAR_MAX` are truncated, so the eventual value stored in `k` (–1) is truncated. To prevent values from being truncated, make sure that you choose a sufficiently wide type that can represent any value that might arise, or check for overflow.

Evaluations

Now that we've looked at simple assignment, let's step back for a moment and look at how expressions are actually evaluated. *Evaluation* mostly means simplifying an expression down to a single value. However, the evaluation of an expression can include both value computations and the initiation of side effects.

A *value computation* is the calculation of the value that results from the evaluation of the expression. Computing the final value may involve determining the identity of the object or reading the value previously assigned to an object. For example, the following expression contains several value computations to determine the identity of `i`, `a`, and `a[i]`:

```
a[i] + f() + 9
```

Because `f` is a function and not an object, the expression `f()` does not involve determining the identity of `f`. The value computations of operands must occur before the value computation of the result of the operator. In this example, separate value computations read the value of `a[i]` and determine the value returned by the call to function `f`. A third computation then sums these values to obtain the value returned by the overall expression. If `a[i]` is an array of `int`, and `f()` returns an `int`, the result of the expression will have the type `int`.

Side effects are changes to the state of the execution environment. Side effects include writing to an object, accessing (reading or writing) a `volatile`-qualified object, I/O, assignment, or calling a function that does any of these things. The previous example can be slightly modified to add an assignment. Updating the stored value of `j` is a side effect of the assignment:

```
int j;
j = a[i] + f() + 9;
```

The assignment to `j` is a side effect that changes the state of the execution environment. Depending on the definition of the `f` function, the call to `f` may also have side effects.

Function Invocation

A *function designator* is an expression that has function type and is used to invoke a function. In the following function invocation, `max` is the function designator:

```
int x = 11;
int y = 21;
int max_of_x_and_y = max(x, y);
```

The `max` function returns the larger of its two arguments. In an expression, a function designator is converted to *pointer-to-function returning type* at compile time. The value of each argument must be of a type that can be assigned to an object with (the unqualified version of) the type of its corresponding parameter. The number and type of the arguments need to agree with the number and type of the parameters accepted by the function. Here, that means two integer arguments. C also supports *variadic*

functions, which can accept a variable number of arguments (the `printf` function is an example of a variadic function).

We can also pass one function to another, as shown by Listing 4-1.

```
int f(void) {
  // ---snip---
  return 0;
}
void g(int (*func)(void)) {
  // ---snip---
  if (func() != 0)
    printf("g failed\n";
  // ---snip---
}
// ---snip---
g(f); // call g with function-pointer argument
// ---snip---
```

Listing 4-1: Passing one function to another function

This code passes the address of a function designated by `f` to another function, `g`. The function `g` accepts a function pointer to a function that accepts no arguments and returns `int`.

Increment and Decrement Operators

The *increment* (`++`) and *decrement* (`--`) operators increment and decrement a modifiable lvalue, respectively. Both are *unary operators*, because they take a single operand.

These operators can be used as either *prefix operators*, which come before the operand, or *postfix operators*, which come after the operand. The prefix and postfix operators have different behaviors, which means they are commonly used as trick questions in quizzes and interviews. A prefix increment performs the increment before returning the value, whereas a postfix increment returns the value and then performs the increment. Listing 4-2 illustrates these behaviors by performing a prefix or postfix increment or decrement operation and then assigning the result to `e`.

```
int i = 5;
int e;    // result of the expression
e = i++;  // postfix increment: i has the value 6; e has the value 5
e = i--;  // postfix decrement: i has the value 5; e has the value 6
e = ++i;  // prefix increment: i has the value 6; e has the value 6
e = --i;  // prefix decrement: i has the value 5; e has the value 5
```

Listing 4-2: Prefix and postfix increment and decrement operators

The `i++` operation in this example returns the unchanged value `5`, which is then assigned to `e`. The value of `i` is then incremented as a side effect of the operation.

The *prefix* increment operator increments the value of the operand, and the expression returns the new value of the operand after it has been incremented. Consequently, the expression `++i` is equivalent to `i = i + 1`, except that `i` is evaluated only once. The `++i` operation in this example returns the incremented value `6`, which is then assigned to `e`.

Operator Precedence and Associativity

In mathematics and computer programming, the *order of operations* (or operator precedence) is a collection of rules that dictate the order in which operations are performed in a given expression. For example, multiplication is normally granted a higher precedence than addition. Therefore, the expression $2 + 3 \times 4$ is interpreted to have the value $2 + (3 \times 4) = 14$, not $(2 + 3) \times 4 = 20$.

Associativity determines how operators of the same precedence are grouped when no explicit parentheses are used. If adjacent operators have equal precedence, the choice of which operation to apply first is determined by the associativity. *Left-associative* operators cause the operations to be grouped from the left while *right-associative* operators cause the operations to be grouped from the right. Grouping can be thought of as the implicit introduction of parentheses. For example, the addition (`+`) operator has left associativity, so the expression `a + b + c` is interpreted as `(a + b) + c`. The assignment operator is right-associative, so the expression `a = b = c` is interpreted as `(a = (b = c))`.

Table 4-1 lists the precedence and associativity of C operators, as specified by the language syntax.[7] Operators are listed top to bottom, in descending precedence.

Table 4-1: Operator Precedence and Associativity

Precedence	Operator	Description	Associativity
0	(…)	Forced grouping	Left
1	`++ --`	Postfix increment and decrement	Left
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Structure and union member access	
	`->`	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal	

7. This table is derived from the C Operator Precedence table at the C++ Reference website, *https://en.cppreference.com/w/c/language/operator_precedence.*

Precedence	Operator	Description	Associativity
2	`++ --`	Prefix increment and decrement	Right
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size of	
	`_Alignof`	Alignment requirement	
3	`* / %`	Multiplication, division, and remainder	Left
4	`+ -`	Addition and subtraction	
5	`<< >>`	Bitwise left shift and right shift	
6	`< <=`	Relational operators < and ≤	
	`> >=`	Relational operators > and ≥	
7	`== !=`	Equal to and not equal to	
8	`&`	Bitwise AND	
9	`^`	Bitwise XOR (exclusive or)	
10	`|`	Bitwise OR (inclusive or)	
11	`&&`	Logical AND	
12	`||`	Logical OR	
13	`?:`	Conditional operator	Right
14	`=`	Simple assignment	
	`+= -=`	Assignment by sum and difference	
	`*= /= %=`	Assignment by product, quotient, and remainder	
	`<<= >>=`	Assignment by bitwise left shift and right shift	
	`&= ^= |=`	Assignment by bitwise AND, XOR, and OR	
15	`,`	Expression sequencing	Left

Sometimes operator precedence can be intuitive, and sometimes it can be misleading. For example, the postfix ++ and -- operators have higher precedence than both the prefix ++ and -- operators, which in turn have the same precedence as the unary * operator. Moreover, if `p` is a pointer, then `*p++` is equivalent to `*(p++)`, and `++*p` is equivalent to `++(*p)`, because both the prefix ++ operator and the unary * operator are right-associative. If two operators have the same precedence and association, they are evaluated from left to right. Listing 4-3 illustrates the precedence rules among these operators.

```
char abc[] = "abc";
char xyz[] = "xyz";

char *p = abc;
```

```
printf("%c", ++*p);

p = xyz;
printf("%c", *p++);
```

Listing 4-3: Operator precedence

The pointer in the expression `++*p` is first dereferenced, producing the character `'a'`. This value is then incremented, resulting in the character `'b'`. On the other hand, the pointer in the expression `*p++` is incremented first, so it refers to the `'y'` character. However, the result of *postfix* increment operators is the value of the operand so that the original pointer value is dereferenced, producing the `'x'` character. Consequently, this code prints out the characters `bx`. You can use parentheses, `( )`, to change or clarify the order of operations.

Order of Evaluation

The *order of evaluation* of the operands of any C operator, including the order of evaluation of any subexpressions, is generally unspecified. The compiler will evaluate them in any order, and may choose a different order when the same expression is evaluated again. This latitude allows the compiler to produce faster code by choosing the most efficient order. The order of evaluation is constrained by operator precedence and associativity.

Listing 4-4 demonstrates the order of evaluation for function arguments. We invoke the `max` function we defined earlier with two arguments, which are the result of calling functions `f` and `g`, respectively. The order of evaluation of the expressions passed to `max` is unspecified, meaning that `f` and `g` could be called in either order.

```
int glob;  // static storage initialized to 0

int f(void) {
  return glob + 10;
}
int g(void) {
  glob = 42;
  return glob;
}
int main(void) {
  int max_value = max(f(), g());
  // ---snip---
}
```

Listing 4-4: Order of evaluation for function arguments

The global variable `glob` is accessed by both functions `f` and `g`, meaning they rely on shared state. When calculating their return value, the values passed as arguments to `max` may differ between compilations. If `f` is called first, it will return 10, but if it is called last, it will return 52. Function `g` always returns 42 regardless of the order of evaluation. Consequently, the

max function (which returns the greater of the two values) may return either 42 or 52, depending on the order of evaluation of its arguments. The only *sequencing guarantees* provided by this code are that both f and g are called before max, and that the executions of f and g do not interleave.

This code can be rewritten to ensure it always behaves in a predictable, portable manner:

```
int f_val = f();
int g_val = g();
int max_value = max(f_val, g_val);
```

In this revised program, f is called to initialize the variable f_val. This is guaranteed to be sequenced before the execution of g, which is called in the subsequent declaration to initialize the variable g_val. If one evaluation is *sequenced before* another evaluation, the first evaluation must complete before the second evaluation can begin. You can use sequence points to guarantee, for example, that an object will be written before it is read as part of a separate evaluation. The execution of f is guaranteed to be sequenced before the execution of g because a sequence point exists between the evaluation of one full expression and the next full expression. Sequence points are discussed in detail in the following subsections.

Unsequenced and Indeterminately Sequenced Evaluations

The executions of unsequenced evaluations can *interleave*, meaning that the instructions can be executed in any order, provided that the execution is *sequentially consistent*—that reads and writes are performed in the order specified by the program (Lamport 1979).

Some evaluations are *indeterminately sequenced*, which means they cannot interleave but can still be executed in any order. For example, the following statement contains several value computations and side effects:

```
printf("%d\n", ++i + ++j * --k);
```

The values of i, j, and k must be read before their values can be incremented or decremented. This means that the reading of i must be sequenced before the increment side effect, for example. Similarly, all side effects for the operands of the multiplication operation need to complete before the multiplication can occur. Finally, the multiplication has to complete before the addition because of operator precedence rules, and all side effects must complete for the operands of the addition operation before it can occur. These constraints produce a partial ordering among these operations, because they don't require that j is incremented before k is decremented, for example. Unsequenced evaluations in this expression can be performed in any order. This allows the compiler to reorder operations and cache values in registers, allowing for faster overall execution. Function executions, on the other hand, are indeterminately sequenced and do not interleave with each other.

Sequence Points

A *sequence point* is the juncture at which all side effects will have completed. These are implicitly defined by the language, but you can control where they occur by the way you specify your program logic.

The sequence points are enumerated in Annex C of the C Standard. A sequence point occurs between the evaluation of one *full expression* (an expression that is not part of another expression or declarator) and the next full expression to be evaluated. A sequence point also occurs upon entering or exiting a called function.

If a side effect is unsequenced relative to either a different side effect on the same scalar or a value computation that uses the value of the same scalar object, the code has undefined behavior. A *scalar type* is either an arithmetic type or pointer type. The expression `i++ * i++` in the following code snippet performs two unsequenced operations on `i`:

```
int i = 5;
printf("Result = %d\n", i++ * i++);
```

You might think this code will produce the value 30, but because this code has undefined behavior, this outcome isn't guaranteed. Conservatively, we can ensure that side effects have completed before the value is read by placing every side-effecting operation in its own full expression. We can rewrite this code as follows to eliminate the undefined behavior:

```
int i = 5;
int j = i++;
int k = i++;
printf("Result = %d\n", j * k);
```

This code now contains a sequence point between every side-effecting operation. However, it's impossible to tell whether this rewritten code represents the original intent of the programmer, because the original code had no defined meaning. If you choose to omit sequence points, you must be sure you completely understand the sequencing of side effects. This same code can also be written as follows without changing the behavior:

```
int i = 5;
int j = i++;
printf("Result = %d\n", j * i++);
```

Now that we have described the mechanics of expressions, we will return to discussing specific operators, starting with the `sizeof` operator.

sizeof Operator

You can use the `sizeof` operator to find the size in bytes of its operand; specifically, it returns an unsigned integer of `size_t` type that represents the size. Knowing the correct size of an operand is necessary for most memory

operations, including allocating and copying storage. The `size_t` type is defined in `<stddef.h>` as well as other header files. You'll need to include one of these header files to compile any code that references `size_t`.

You can pass the `sizeof` operator an unevaluated expression of a complete object type or a parenthesized name of such a type:

```
int i;
size_t i_size = sizeof i;      // the size of the object i
size_t int_size = sizeof(int); // the size of the type int
```

It is always safe to parenthesize the operand to `sizeof`, because parenthesizing an expression does not change the way the size of the operand is calculated. The result of invoking the `sizeof` operator is a constant expression unless the operand is a variable-length array. The operand to `sizeof` is not evaluated.

If you need to determine the number of bits of storage available, you can multiply the size of an object by `CHAR_BIT`, which gives the number of bits contained in a byte. For example, the expression `CHAR_BIT * sizeof(int)` will produce the number of bits in an object of type `int`.

Object types other than character types can include padding as well as value representation bits. Different target platforms can pack bytes into multiple-byte words in different ways, called *endianness*.[8] All this variation implies that, for interhost communication, you should adopt a standard for the external format and use format conversion functions to *marshal* arrays of external data to and from multiple-byte native objects.

Arithmetic Operators

Several operators that perform arithmetic operations on arithmetic types are detailed in the following sections. Some of these operators can also be used with nonarithmetic operands.

Unary + and – Operators

The *unary + and - operators* operate on a single operand of arithmetic type. The - operator returns the negative of its operand (that is, it behaves as though the operand were multiplied by –1). The unary + operator just returns the value. These operators exist primarily to express positive and negative numbers.

If the operand has a small integer type, it is promoted (see Chapter 3), and the result of the operation has the result of the promoted type. As a point of trivia, C has no negative integer constants; a value such as -25 is actually an rvalue of type `int` with the value 25 preceded by the unary - operator.

8. This term is drawn from Jonathan Swift's 1726 satire, *Gulliver's Travels*, in which civil war erupts over whether the big end or the little end of a boiled egg is the proper end to crack open.

Logical Negation Operator

The result of the unary *logical negation* (`!`) *operator* is as follows:

- 0 if the value of its operand is not 0
- 1 if the value of its operand is 0

The operand is a scalar type. The result has type `int` for historical reasons. The expression `!E` is equivalent to `(0 == E)`. The logical negation operator is frequently used to check for null pointers; for example, `!p` is equivalent to `(NULL == p)`.

Multiplicative Operators

The binary *multiplicative operators* include multiplication (`*`), division (`/`), and remainder (`%`). The usual arithmetic conversions are implicitly performed on multiplicative operands to find a common type. You can multiply and divide both floating-point and integer operands, but remainder operates on only integer operands.

Various programming languages implement different kinds of integer division operations, including Euclidean, flooring, and truncating. In *Euclidean division*, the remainder is always nonnegative (Boute 1992). In *flooring division*, the quotient is rounded toward negative infinity (Knuth 1997). In *truncating division*, the result of the `/` operator is the algebraic quotient with any fractional part discarded. This is often referred to as *truncation toward zero*.

The C programming language implements *truncating division*, meaning that the remainder always has the same sign as the dividend, as shown in Table 4-2.

Table 4-2: Truncating Division

`/`	Quotient	`%`	Remainder
`10 / 3`	3	`10 % 3`	1
`10 / –3`	–3	`10 % –3`	1
`–10 / 3`	–3	`–10 % 3`	–1
`–10 / –3`	3	`–10 % –3`	–1

To generalize, if the quotient `a / b` is representable, then the expression `(a / b) * b + a % b` equals `a`. Otherwise, if the value of the divisor is equal to 0 or `a / b` overflows, both `a / b` and `a % b` will result in undefined behavior.

It's worth taking the time to understand the behavior of the `%` operator to avoid surprises. For example, the following code defines a faulty function called `is_odd` that attempts to test whether an integer is odd:

```
bool is_odd(int n) {
  return n % 2 == 1;
}
```

Because the result of the remainder operation always has the sign of the dividend `n`, when `n` is negative and odd, `n % 2` returns `-1`, and the function returns `false`.

A correct, alternative solution is to test that the remainder is not 0 (because a remainder of 0 is the same regardless of the sign of the dividend):

```
bool is_odd(int n) {
  return n % 2 != 0;
}
```

Many CPUs implement remainder as part of the division operator, which can overflow if the dividend is equal to the minimum negative value for the signed integer type and the divisor is equal to –1. This occurs even though the mathematical result of such a remainder operation is 0.

The C Standard Library provides floating-point remainder, truncation, and rounding functions, including `fmod`.

Additive Operators

The binary *additive operators* include addition (`+`) and subtraction (`-`). Addition and subtraction can be applied to two operands of arithmetic types, but you can also use them to perform scaled pointer arithmetic. I'll discuss pointer arithmetic near the end of this chapter; the discussion here is limited to operations on arithmetic types.

The binary `+` operator sums its two operands. The binary `-` operator subtracts the right operand from the left operand. The usual arithmetic conversions are performed on operands of arithmetic type for both operations.

Bitwise Operators

We use *bitwise operators* to manipulate the bits of an object or any integer expression. Typically, they're used on objects that represent *bitmaps*: each bit indicates that something is "on" or "off," "enabled" or "disabled," or another binary pairing.

Bitwise (`| & ^ ~`) operators treat the bits as a pure binary model without concern for the values represented by these bits:

```
  1 1 0 1 = 13
^ 0 1 1 0 = 6
= 1 0 1 1 = 11
```

Bitmaps are best represented as unsigned integer types, as the sign bit can be better used as a value within the bitmap, and operations on the values are less prone to undefined behavior.

Complement Operator

The *unary complement* (~) *operator* works on a single operand of integer type and returns the *bitwise complement* of its operand; that is, a value in which each bit of the original value is flipped. The complement operator is used in applying the POSIX `umask`, for example. A file's permission mode is the result of a logical AND operation between the complement of the mask and the process's requested permission mode setting. Integer promotions are performed on the operand, and the result has the promoted type. For example, the following code snippet applies the ~ operator to a value of type `unsigned char`:

```
unsigned char uc = UCHAR_MAX; // 0xFF
int i = ~uc;
```

On an architecture with an 8-bit `char` type and 32-bit `int` type, `uc` is assigned the value `0xFF`. When `uc` is used as the operand to the ~ operator, `uc` is promoted to `signed int` by zero-extending it to 32 bits, `0x000000FF`. The complement of this value is `0xFFFFFF00`. Therefore, on this platform, complementing an `unsigned short` type always results in a negative value of type `signed int`. As a general policy and to avoid surprises such as this, all bitwise manipulations should involve a single, sufficiently wide unsigned integer type.

Shift Operators

Shift operations shift the value of each bit of an operand of integer type by a specified number of positions. Shifting is commonly performed in systems programming, where bit masks are common. They may also be used in code that manages network protocols or file formats to pack or unpack data. They include left-shift operations, of the form:

```
shift-expression << additive-expression
```

and right-shift operations, of the form:

```
shift expression >> additive expression
```

The *shift expression* is the value to be shifted, and the *additive expression* is the number of bits by which to shift the value. Figure 4-1 illustrates a logical left shift of 1 bit.

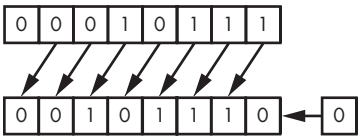

Figure 4-1: Logical left shift of 1 bit

The additive expression determines the number of bits by which to shift the value. For example, the result of `E1 << E2` is the value of `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has an unsigned type,

the value of the result is `E1 × 2^E2`. If the value cannot be represented in the resulting type, it is reduced modulo one more than the maximum representable value. If `E1` has a signed type and nonnegative value, and if `E1 × 2^E2` is representable in the result type, then that is the resulting value; otherwise, it is undefined behavior. Similarly, the result of `E1 >> E2` is the value of `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of `E1/2^E2`. If `E1` has a signed type and a negative value, the resulting value is implementation defined and may be either an arithmetic (sign-extended) shift or a logical (unsigned) shift, as shown in Figure 4-2.

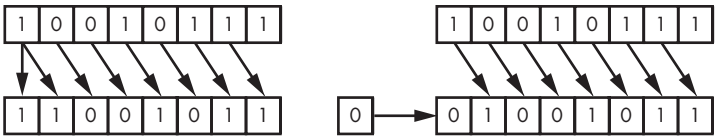

Figure 4-2: Arithmetic (signed) right shift and logical (unsigned) right shift of 1 bit

In both kinds of shifts, the integer promotions are performed on the operands, each of which has the integer type. The type of the result is that of the promoted left operand. The usual arithmetic conversions are *not* performed.

Even though you could use left and right shifts to multiply or divide by a power of two, it's not a good idea to use a shift operation for this purpose. It is best to use the multiplication and division operations and let the compiler determine when to optimize processing by replacing these operations with a shift operation. Performing this substitution yourself is an example of premature optimization. Donald Knuth, author of *The Art of Computer Programming* (1997), has described premature optimization as "the root of all evil."

The number of bits shifted should not be negative, greater than, or equal to the width of the promoted left operand, because doing so is undefined behavior. Listing 4-5 shows how to perform right-shift operations on signed and unsigned integers that are free from these errors.

```
extern int si1, si2, sresult;
extern unsigned int ui1, ui2, uresult;
// ---snip---
❶ if ( (si2 < 0) || (si2 >= sizeof(int)*CHAR_BIT) ) {
    /* error */
  }
  else {
    sresult = si1 >> si2;
  }
❷ if (ui2 >= sizeof(unsigned int)*CHAR_BIT) {
    /* error */
  }
  else {
    uresult = ui1 >> ui2;
  }
```

Listing 4-5: Correct right-shift operations

For signed integers, ❶ you must ensure that the number of bits shifted is not negative, or greater than, or equal to the width of the promoted left operand. For unsigned integers, ❷ you omit the test for negative values, as unsigned integers can never be negative. You can perform left-shift operations in a similar manner.

Bitwise AND Operator

The *binary bitwise AND* (`&`) *operator* returns the bitwise AND of two operands of integer type. The usual arithmetic conversions are performed on both operands. Each bit in the result is set if and only if each of the corresponding bits in the converted operands is set, as shown in Table 4-3.

Table 4-3: Bitwise AND Truth Table

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise Exclusive OR Operator

The *bitwise exclusive OR* (`^`) *operator* returns the bitwise exclusive OR of the operands of integer type. In other words, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set, as shown in Table 4-4. This is sometimes thought of as "one or the other, but not both."

Table 4-4: Bitwise Exclusive OR Truth Table

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive OR is equivalent to the addition operation on the integers modulo 2—that is, because of wraparound 1 + 1 mod 2 = 0 (Lewin 2012). The operands must be integers, and the usual arithmetic conversions are performed on both.

Beginners commonly mistake the exclusive OR operator for an exponent operator, erroneously believing that the expression `2 ^ 7` will compute 2 raised to the power of 7. The correct way to raise a number to a certain power in C is to use the `pow` functions defined in `<math.h>`, as shown in

Listing 4-6. The pow functions operate on floating arguments and return a floating result, so be aware that these functions might fail to produce the expected results because of truncation or other errors.

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

int main(void) {
  int i = 128;
  if (i == pow(2, 7)) {
    puts("equal");
  }
}
```

Listing 4-6: Using the pow functions

This code calls the pow function to compute 2 raised to the power of 7. Because 2^7 equals 128, and assuming 128 is exactly representable in the type double, this program will print equal.

Bitwise Inclusive OR Operator

The *bitwise inclusive OR* (|) *operator* operator returns the bitwise inclusive OR of the operands of integer type. The operands must be integers, and the usual arithmetic conversions are performed on both. Each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set, as shown in Table 4-5.

Table 4-5: Bitwise Inclusive OR Truth Table

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Logical Operators

The *logical AND* (&&) *operator* and *logical* (||) *OR operator* are used primarily for logically joining two or more expressions of scalar type. They're commonly used in condition tests, such as in the first operand of the conditional operator, the controlling expression of an if statement, or the controlling expression of a for loop, to combine multiple comparisons together. You shouldn't use logical operators with bitmap operands, as they are intended primarily for Boolean logic.

The && operator returns 1 if neither of its operands is equal to 0, and returns 0 otherwise. Logically, this means that a && b is true only if both a is true and b is true.

The || operator returns 1 if either of its operands is not equal to 0, and returns 0 otherwise. Logically, this means that `a || b` is true if `a` is true, `b` is true, or both `a` and `b` are true.

The C Standard defines both operations in terms of "not equal to zero" because the operands can have values other than 0 and 1. Both operators accept operands of scalar type (integers, floats, and pointers), and the result of the operation has type `int`.

Unlike the corresponding bitwise binary operators, the logical AND operator and logical OR operator guarantee left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands.

The logical operators *short-circuit*: the second operand is not evaluated if the result can be deduced solely by evaluating the first operand. For example, the expression `0 && unevaluated` returns 0 regardless of the value of `unevaluated` because there is no possible value for `unevaluated` that produces a different result. Because of this, `unevaluated` is not evaluated to determine its value. The same is true for `1 || unevaluated` because this expression will always return 1.

Short-circuiting is commonly used in operations with pointers:

```
bool isN(int* ptr, int n){
  return ptr && *ptr == n; // don't dereference a null pointer
}
```

This code tests the value of `ptr`. If `ptr` is `NULL`, the second `&&` operand is not evaluated, preventing a null pointer dereference.

This behavior is a useful way to avoid unnecessary computing. Here, the `is_file_ready` predicate function will return `true` if the file is ready:

```
is_file_ready() || prepare_file()
```

In this case, when the `is_file_ready` function returns `true`, the second || operand is not evaluated, as there is no need to prepare the file. This will avoid unnecessary computing, assuming the cost of determining whether the file is ready is less than the cost of preparing the file (and there's a good chance the file is already prepared).

Programmers should exercise caution if the second operand contains side effects, because it may not be apparent whether the side effects actually occur. For example, in the following code snippet, the value of `i` is incremented only when `i >= 0`:

```
enum { max = 15 };
int i = 17;

if ( (i >= 0) && ( (i++) <= max) ) {
  // ---snip---
}
```

This code may be correct, but it's more likely that it's an error.

Cast Operators

Casts (also known as *type casts*) explicitly convert a value of one type to a value of another type. To perform a cast, we precede an expression by a parenthesized type name, which converts the value of the expression to the unqualified version of the named type. The following code illustrates an explicit conversion, or cast, of `x` from type `double` to type `int`:

```
double x = 1.2;
int sum = (int)x + 1;  // Explicit conversion from double to int
```

Unless the type name specifies a `void` type, the type name must be a qualified or unqualified scalar type. The operand must also have scalar type; a pointer type cannot be converted to any floating-point type, and vice versa.

Casts are extremely powerful and must be used carefully. For one thing, casts may reinterpret the existing bits as a value of the specified type without changing the bits:

```
intptr_t i = (intptr_t)some_pointer; // reinterpret bits as an integer
```

Casts may also change these bits into whatever bits are needed to represent the original value in the resulting type:

```
int i = (int)some_float; // change bits to an integer representation
```

Casts can also disable diagnostics. For example, consider the following code snippet:

```
char c;
// ---snip---
while ((c = fgetc(in)) != EOF) {
  // ---snip---
}
```

This generates the following diagnostic when compiled with Visual C++ 2019 with warning level `/W4`:

```
Severity  Code   Description
Warning   C4244  '=': conversion from 'int' to 'char', possible loss of data
```

Adding a cast to `char` disables the diagnostic without fixing the problem:

```
char c;
while ((c = (char)fgetc(in)) != EOF) {
  // ---snip---
}
```

To mitigate these risks, C++ defines its own casts, which are less powerful.

Conditional Operator

The *conditional* (`?:`) *operator* is the only C operator that takes three operands. It returns a result based on the condition. You can use the conditional operator like this:

```
result = condition ? valueReturnedIfTrue : valueReturnedIfFalse;
```

The conditional operator evaluates the first operand, called the *condition*. It evaluates either the second operand (`valueReturnedIfTrue`) if the condition is true, or the third operand (`valueReturnedIfFalse`) if the condition is false. The result is the value of either the second or third operand (depending on which operand was evaluated).

This result is converted to a common type based on the second and third operands. There is a sequence point between the evaluation of the first operand and the evaluation of the second or third operand (whichever is evaluated) so that the compiler will ensure that all side effects have completed before the second or third operand is evaluated.

The conditional operator is similar to an `if-else` control flow block, but returns a value as a function does. Unlike with an `if-else` control flow block, you can use the conditional operator to initialize a `const`-qualified object:

```
const int x = (a < b) ? b : a;
```

The first operand to the conditional operator must have scalar type. The second and third operands must have compatible types (roughly speaking). For more details on the constraints for this operator and the specifics of determining the return type, refer to Section 6.5.15 of the C Standard (ISO/IEC 9899:2018).

_Alignof Operator

The `_Alignof` operator yields an integer constant representing the alignment requirement of its operand's declared complete object type. It does not evaluate the operand. When applied to an `array` type, it returns the alignment requirement of the element type. This operator is typically used through the convenience macro `alignof`, which is provided in the header `<stdalign.h>`. The `_Alignof` operator is useful in both static and runtime assertions that are used to verify assumptions about your program (discussed further in Chapter 11). The purpose of these assertions is to diagnose situations in which your assumptions are invalid. Listing 4-7 demonstrates the use of the `_Alignof` operator and the `alignof` macro.

```
#include <stdio.h>
#include <stddef.h>
#include <stdalign.h>
#include <assert.h>

int main(void) {
  int arr[4];
  static_assert(_Alignof(arr) == 4, "unexpected alignment"); // static assert
  assert(alignof(max_align_t) == 16); // runtime assertion
  printf("Alignment of arr = %zu\n", _Alignof(arr));
  printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
}
```

Listing 4-7: Using the `_Alignof` operator

This simple program doesn't accomplish anything particularly useful. It declares an array `arr` of four integers followed by a static assertion concerning the alignment of the array, and a runtime assertion concerning the alignment of `max_align_t` (an object type whose alignment is the greatest fundamental alignment). It then prints out these values. If runtime assertions are enabled, this program may fail to compile because of the `static_assert`, fail the runtime assertion and not print anything, or output the following:

```
Alignment of arr = 4
Alignment of max_align_t = 16
```

Relational Operators

The *relational operators* include `==` (equal to), `!=` (not equal to), `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to). Each returns 1 if the specified relationship is true, and 0 if it is false. The result has type `int`, again, for historical reasons.

Note that C does not interpret the expression `a < b < c` to mean that `b` is greater than `a` but less than `c`, as ordinary mathematics does. Instead, the expression is interpreted to mean `(a < b) < c`. In English, if `a` is less than `b`, the compiler should compare `1` to `c`; otherwise, it compares `0` to `c`. If this is your intent, you should include the parentheses to make this clear to any potential code reviewer. Some compilers such as GCC and Clang provide the `-Wparentheses` flag that diagnoses these problems. A test to determine whether `b` is greater than `a` but less than `c` can be written as `(a < b) && (b < c)`.

The equality and inequality operators have lower precedence than the relational operators—and assuming otherwise is a common mistake. This means that the expression `a < b == c < d` is evaluated the same as `(a < b) == (c < d)`. In both cases, the comparisons `a < b` and `c < d` are evaluated first, and the resulting values (either `0` or `1`) are compared for equality.

We can use these operators to compare arithmetic types or pointers. When we compare two pointers, the result depends on the relative locations in the address space of the objects pointed to. If both pointers point to the same object, they are equal.

The equality and inequality operators differ from the other relational operators. For example, you cannot use the other relational operators on two pointers to unrelated objects, because doing so makes no sense:

```
int i, j;
bool b1 = &i < &j;  // undefined behavior
bool b2 = &i == &j; // okay, but tautologically false
```

Compound Assignment Operators

Compound assignment operators modify the current value of an object by performing an operation on it. The compound assignment operators are shown in Table 4-6.

Table 4-6: Compound Assignment Operators

Operator	Description
`+= -=`	Assignment by sum and difference
`*= /= %=`	Assignment by product, quotient, and remainder
`<<= >>=`	Assignment by bitwise left shift and right shift
`&= ^= |=`	Assignment by bitwise AND, XOR, and OR

A compound assignment of the form `E1` *op* `= E2` is equivalent to the simple assignment expression `E1 = E1` *op* `(E2)`, except that `E1` is evaluated only once. Compound assignments are primarily used as a shorthand notation. There are no compound assignment operators for logical operators.

Comma Operator

In C, we use commas in two distinct ways: as operators and as a way to separate items in a list (such as arguments to functions or lists of declarations). The *comma* (`,`) *operator* is a way to evaluate one expression before another. First, the left operand of a comma operator is evaluated as a `void` expression. There is a sequence point between the evaluation of the left operand and the evaluation of the right operand. Then, the right operand is evaluated after the left. The result of the comma operation has the type and value of the right operand—mostly because it is the last expression evaluated.

You cannot use the comma operator in contexts in which a comma might separate items in a list. Instead, you would include a comma within

a parenthesized expression, or within the second expression of a conditional operator. For example, the following function call has three parameters:

```
f(a,❶ (t=3,❷ t+2),❸ c)
```

The first comma ❶ separates the first and second arguments to the function. The second comma ❷ is a comma operator. The assignment is evaluated first, followed by the addition. Because of the sequence point, the assignment is guaranteed to complete before the addition takes place. The result of the operation has the type (`int`) and value (`5`) of the right-hand operand. The third comma ❸ separates the second and third arguments to the function.

Pointer Arithmetic

Earlier in this chapter, we mentioned that it's possible to use the additive operators (addition and subtraction) on either arithmetic values or object pointers. In this section, we discuss adding a pointer and an integer, subtracting two pointers, and subtracting an integer from a pointer.

Adding or subtracting an expression that has integer type to or from a pointer returns a value with the type of the pointer operand. If the pointer operand points to an element of an array, then the result points to an element offset from the original element. If the resulting pointer is outside the bounds of the array, undefined behavior occurs. The difference of the array subscripts of the resulting and original array elements equals the integer expression:

```
int arr[100];
int *arrp1 = arr[40];
int *arrp2 = arrp1 + 20;      // arrp2 points to arr[60]
printf("%td\n", arrp2-arrp1); // prints 20
```

C allows a pointer to be formed to each element of an array, including one past the last element of the array object (also referred to as the *too-far* pointer). While this might seem unusual or unnecessary, many early C programs incremented a pointer until it was equal to the too-far pointer, and the C Standards committee did not want to break all this code, which is also idiomatic in C++ iterators. Figure 4-3 illustrates forming the *too-far* pointer.

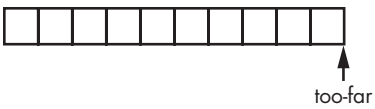

Figure 4-3: One past the last element of an array object

If both the pointer operand and the result point to elements of the same array object or the too-far pointer, the evaluation will not overflow;

otherwise, the behavior is undefined. To satisfy the too-far requirement, an implementation need only provide one extra byte (which can overlap another object in the program) just after the end of the object.

C also allows objects to be treated as an array containing only a single element, allowing you to obtain a too-far pointer from a scalar.

The too-far special case allows us to advance a pointer until it is equal to the too-far pointer, as in the following function:

```
int m[2] = {1, 2};

int sum_m_elems(void) {
  int *pi; int j = 0;
  for (pi = &m[0]; pi < &m[2]; ++pi) j += *pi;
  return j;
}
```

Here, the `for` statement (explained in detail in the next chapter) in the `sum_m_elems` function loops while `pi` is less than the address of the too-far pointer for the array `m`. The pointer `pi` is incremented at the end of each iteration of the loop until the too-far pointer is formed, causing the loop condition to subsequently evaluate to 0.

When we subtract one pointer from another, both must point to elements of the same array object or the too-far element. This operation returns the difference of the subscripts of the two array elements. The type of the result is `ptrdiff_t` (a signed integer type). You should take care when subtracting pointers, because the range of `ptrdiff_t` may not be sufficient to represent the difference of pointers to elements of very large arrays of `char`. Pointer arithmetic is automatically *scaled* to work with array element sizes, rather than individual bytes.

Summary

In this chapter, you learned how to use operators to write simple expressions that perform operations on various object types. Along the way, you learned about some core C concepts, such as lvalues, rvalues, value computations, and side effects, which determine how expressions are evaluated. You also learned how operator precedence, associativity, order of evaluation, sequencing, and interleaving can affect the total order in which a program is executed. In the next chapter, you'll learn more about how to control the execution of your program by using selection, iteration, and jump statements.

5

CONTROL FLOW

In this chapter, you'll learn how to control the order in which individual statements are evaluated. We'll start by going over expression statements and compound statements that define the work to be performed. We'll then cover three kinds of statements that determine which code blocks are executed, and in what order: selection, iteration, and jump statements.

Expression Statements

An *expression statement* is an optional expression terminated by a semicolon. It is one of the most common kinds of statements and a basic unit of work. Listing 5-1 shows examples of expression statements.

```
a = 6;
c = a + b;
;  // null statement, does nothing
++count;
```

Listing 5-1: Typical expression statements

The first statement consists of an expression that assigns a value to `a`. The second statement is an expression that assigns the sum of `a` and `b` to `c`. The third statement is a null statement, which you can use when the syntax of the language requires a statement but no expression needs to be evaluated. Null statements are commonly used as placeholders in iteration statements or as statements on which to place labels at the end of compound statements or functions. The fourth statement is an expression that increments the value of `count`.

After each full expression has been evaluated, its value (if any) is discarded (including assignment expressions in which the assignment itself is a side effect of the operation) so that any useful results occur as the consequence of side effects (as discussed in Chapter 4). Three of the four expression statements in this example have side effects (the null statement does nothing). Once all side effects have completed, execution proceeds to the statement following the semicolon.

Compound Statements

A *compound statement*, or *block*, is a list of zero or more statements, surrounded by braces. The statements in the block may be any kind of statement described throughout this chapter. Some of these statements may be declarations. (In early versions of C, declarations within the block had to precede all nondeclarations, but that restriction no longer applies.) Each statement in the block is executed in sequence, unless modified by a control statement. After the final statement has been evaluated, execution proceeds to just after the closing brace:

```
{
  static int count = 0;
  c += a;
  ++count;
}
```

This example declares a static variable of type `int` called `count`. The second line increases a variable `c` declared in an outer scope by the value stored in `a`. Finally, `count` is incremented to track how many times this block has been executed.

Compound statements can be nested so that one compound statement fully encloses another. You may also have blocks with no statements at all (just the empty braces).

CODE STYLE

Competing coding styles disagree on when and where to place braces. If you're modifying existing code, it would be wise to follow the style already in use for the project. Otherwise, take a look at styles you see in code written by experienced C programmers and choose one that seems clear. For example, some programmers line up the opening and closing braces to make it easy to find the mate for a given brace. Others follow the style used in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (1988), wherein the opening brace is placed at the end of the preceding line, and the closing brace gets a line to itself. Once you have chosen a style, use it consistently.

Selection Statements

Selection statements allow you to conditionally execute a substatement depending on the value of a controlling expression. The *controlling expression* determines which statements are executed based on a condition. This allows you to write code to produce different output based on different inputs. Selection statements include the `if` statement and the `switch` statement.

The if Statement

The `if` statement allows a programmer to execute a substatement based on the value of a controlling expression of scalar type.

There are two kinds of `if` statements. The first conditionally determines whether the substatement is executed:

```
if (expression)
  substatement
```

In this case, *`substatement`* is executed if *`expression`* is not equal to 0. Only the single substatement of the `if` statement is conditionally executed, though it can be a compound statement.

Listing 5-2 shows a division function that uses `if` statements. It divides a specified dividend by a specified divisor and returns the result in the object referenced by `quotient`. The function tests for both division by zero and signed integer overflow, and returns `false` in either case.

```
bool safediv(int dividend, int divisor, int *quotient) {
❶ if (!quotient) return false;
❷ if ((divisor == 0) || ((dividend == INT_MIN) && (divisor == -1)))
  ❸ return false;
❹ *quotient = dividend / divisor;
  return true;
}
```

Listing 5-2: Safe division function

The first line of this function ❶ tests `quotient` to ensure that it's not null. If it is null, the function returns `false` to indicate that it is unable to return a value. We cover `return` statements later in this chapter.

The second line of the function ❷ contains a more complex `if` statement. Its controlling expression tests whether the divisor is 0 or whether the division would result in signed integer overflow if unchecked. If the result of this expression is not equal to 0, the function returns `false` ❸ to indicate that it is unable to produce a quotient. If the controlling expression of the `if` statement evaluates to 0, the function does not return, and the remaining statements ❹ are executed.

The second kind of `if` statement includes an `else` clause, which selects an alternative substatement to execute when the initial substatement is not selected:

```
if (expression)
  substatement1
else
  substatement2
```

In this form, *`substatement1`* is executed if *`expression`* is not equal to 0, and *`substatement2`* is executed if the expression is equal to 0. It is always the case that one or the other of these substatements is executed, but never both.

For either form of the `if` statement, the conditionally executed substatement may also be an `if` statement. A common use of this is the `if...else` ladder, shown in Listing 5-3.

```
if (expr1)
  substatement1
else if (expr2)
  substatement2
else if (expr3)
  substatement3
else
  substatement4
```

Listing 5-3: `if...else` ladder

One (and only one) of the four statements in an `if...else` ladder will execute:

- *`substatement1`* executes if *`expr1`* does not equal 0.
- *`substatement2`* executes if *`expr1`* equals 0 and if *`expr2`* does not equal 0.
- *`substatement3`* executes if both *`expr1`* and *`expr2`* equal 0 and *`expr3`* does not equal 0.
- *`substatement4`* executes only if the proceeding conditions are all equal to 0.

The example shown in Listing 5-4 uses an `if..else` ladder to print grades.

```
void printgrade(unsigned int marks) {
  if (marks >= 90) {
    puts("YOUR GRADE : A");
  } else if (marks >= 80) {
    puts("YOUR GRADE : B");
  } else if (marks >= 70) {
    puts("YOUR GRADE : C");
  } else {
    puts("YOUR GRADE : Failed");
  }
}
```

Listing 5-4: Using an `if...else` ladder to print grades

In this `if...else` ladder, the `printgrade` function tests the value of the `unsigned int` parameter `marks` to determine whether it is greater than or equal to 90. If so, the function prints `YOUR GRADE : A`. Otherwise, it tests whether `marks` is greater than or equal to 80, and so forth down the `if...else` ladder. If `marks` is not greater than or equal to 70, the function prints `YOUR GRADE : Failed`. This example uses a coding style in which the closing brace is followed by the `else` clause on the same line.

Only a single statement following the `if` statement is executed. For example, in the following code snippet, `conditionally_executed_function` is executed only if `condition` is not equal to 0, but `unconditionally_executed_function` is always executed:

```
  if (condition)
    conditionally_executed_function();
  unconditionally_executed_function(); // always executed
```

Attempting to add another conditionally executed function is a common source of errors:

```
  if (condition)
    conditionally_executed_function();
    second_conditionally_executed_function();  // ????
  unconditionally_executed_function(); // always executed
```

In this code snippet, `second_conditionally_executed_function` is *unconditionally* executed. The name and indented formatting are deceptive, because whitespace (in general) and indentation (in particular) are meaningless to the syntax. This code can be fixed by adding braces, delimiting a single compound statement or block, which is then executed as the single conditionally executed statement:

```
  if (condition) {
    conditionally_executed_function();
    second_conditionally_executed_function(); // fixed it
  }
  unconditionally_executed_function(); // always executed
```

While the original code snippet was not incorrect, many coding guidelines recommend always including braces to avoid this kind of error:

```
if (condition) {
  conditionally_executed_function();
}
unconditionally_executed_function(); // always executed
```

My personal style is to omit the braces only when I can include the conditionally executed statement on the same line as the `if` statement:

```
if (!quotient) return false;
```

This issue is less problematic when you let your IDE format your code for you, as it will not be fooled by the presence or absence of braces when formatting your code. Some compilers, such as GCC's `-Wmisleading-indentation`, also check code indentation and issue warnings when it doesn't correspond to the control flow.

The switch Statement

The `switch` statement works just like the `if...else` ladder, except that the controlling expression must have an integer type. For example, the `switch` statement in Listing 5-5 performs the same function as the `if...else` ladder from Listing 5-4, provided that `marks` is an integer in the range of 0 to 100. If `marks` is greater than 109, it will result in a failed grade because the resulting quotient will be greater than 10 and will consequently be caught by the default case.

```
switch (marks/10) {
  case 10:
  case 9:
    puts("YOUR GRADE : A");
    break;
  case 8:
    puts("YOUR GRADE : B");
    break;
  case 7:
    puts("YOUR GRADE : C");
    break;
  default:
    puts("YOUR GRADE : Failed");
}
```

Listing 5-5: Using a `switch` statement to print out grades

The `switch` statement will cause control to jump to one of the three substatements, depending on the value of the controlling expression and the constant expressions in each `case` label. Following the jump, code is executed sequentially until the next control flow statement is reached. In our

example, a jump to `case 10` (which is empty) will flow through and execute the subsequent statements in `case 9`. This is necessary to the logic so that a perfect grade of 100 results in an A and not an F.

You can terminate the execution of the `switch`, causing control to jump to the execution of the statement directly following the overall `switch` statement. We discuss `break` statements in more detail later in this chapter. Make sure you remember to include a `break` statement before the next `case` label. If omitted, the control flow will fall through to the next `case` in the `switch` statement—a common source of errors. Because the `break` statement is not required, omitting it doesn't typically produce compiler diagnostics. GCC will issue a warning for fall-through if you use the `-Wimplicit-fallthrough` flag. The C2x standard will introduce a `[[fallthrough]]` attribute as a way for a programmer to specify that fall-through behavior is desirable, under the assumption that silent fall-through is an accidental omission of a `break` statement.

Integer promotions are performed on the controlling expression. The constant expression in each `case` label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched `case` label. Otherwise, if there is a `default` label, control jumps to the labeled statement. If no converted `case` constant expression matches and there is no `default` label, no part of the switch body is executed. When `switch` statements are nested, a `case` or `default` label is accessible only within the closest enclosing `switch` statement.

There are best practices regarding the use of `switch` statements. Listing 5-6 shows a *not incorrect* implementation of a `switch` statement that assigns interest rates to an account based on the account type. A fixed number of account types are offered by this bank, so the account types are represented by the `AccountType` enumeration.

```
typedef enum { Savings, Checking, MoneyMarket } AccountType;
void assignInterestRate(AccountType account) {
  double interest_rate;
  switch (account) {
    case Savings:
      interest_rate = 3.0;
      break;
    case Checking:
      interest_rate = 1.0;
      break;
    case MoneyMarket:
      interest_rate = 4.5;
      break;
  }
  printf("Interest rate = %g.\n", interest_rate);

}
```

Listing 5-6: `switch` statement without a `default` label

The `assignInterestRate` function defines a single parameter of the enumeration type `assignInterestRate` and switches on it to assign the appropriate interest rate associated with each account type. Nothing is wrong with the code as written, but it requires programmers to update the code in at least two separate places if they want to make any changes. Let's say the bank introduces a new type of account: a certificate of deposit. The programmers update the `AccountType` enumeration as follows:

```
typedef enum { Savings, Checking, MoneyMarket, CD } AccountType;
```

However, they neglect to modify the `switch` statement in the `assignInterestRate` function. Therefore, `interest_rate` is not assigned, resulting in an uninitialized read when the function attempts to print this value. This is a common problem because the enumeration may be declared far from the `switch` statement, and the program may contain many similar `switch` statements that all reference an object of type `AccountType` in their controlling expression. Both Clang and GCC will help diagnose these problems at compilation time when you use the `-Wswitch-enum` flag. Alternatively, you can protect against such errors by including a `default` label in the `switch`:

```
    default: abort();
```

The `abort` function (declared in the `<stdlib.h>` header) causes abnormal program termination, making it easy to detect the error. You can improve the testability of this code by adding this `default` case to the `switch` statement, as shown in Listing 5-7.

```
typedef enum { Savings, Checking, MoneyMarket, CD } AccountType;
void assignInterestRate(AccountType account) {
  double interest_rate;
  switch (account) {
    case Savings:
      interest_rate = 3.0;
      break;
    case Checking:
      interest_rate = 1.0;
      break;
    case MoneyMarket:
      interest_rate = 4.5;
      break;
    case CD:
      interest_rate = 7.5;
      break;
    default: abort();
  }
  printf("Interest rate = %g.\n", interest_rate);
  return;
}
```

Listing 5-7: `switch` statement with a `default` label

The `switch` statement now includes a case for `CD`, and the `default` clause is unused. However, retaining the `default` clause is good practice, in case another account type is added in the future.

Including a `default` clause does have the drawback of suppressing compiler warnings and not diagnosing the problem until runtime. Compiler warnings (if supported by your compiler) are therefore a better approach.

Iteration Statements

Iteration statements cause substatements (or compound statements) to be executed zero or more times, subject to termination criteria. *Iteration* is an English word meaning "the repetition of a process." Iteration statements are more informally and commonly referred to as loops. *Loop* is also an English word meaning "a process, the end of which is connected to the beginning."

The while Statement

The `while` statement causes the loop body to execute repeatedly until the controlling expression is equal to 0. The evaluation of the controlling expression occurs before each execution of the loop body. Consider the following example:

```
void f(unsigned int x) {
  while (x > 0) {
    printf("%d\n," x);
    --x;
  }
  return;
}
```

If `x` is not initially greater than 0, the `while` loop exits without executing the loop body. If `x` is greater than 0, its value is output and then decremented. Once the end of the loop is reached, the controlling expression is tested again. This pattern repeats until the expression evaluates to 0. Overall, this loop will count down from `x` to 1.

A `while` loop is a simple entry-controlled loop that executes for as long as its entry condition is met. Listing 5-8 shows an implementation of the C Standard Library `memset` function. This function copies the value of `val` (converted to an `unsigned char`) into each of the first `n` characters of the object pointed to by `dest`.

```
void *memset(void *dest, int val, size_t n) {
  unsigned char *ptr = (unsigned char*)dest;
  while (n-- > 0)
    *ptr++ = (unsigned char)val;
  return dest;
}
```

Listing 5-8: The C Standard Library `memset` function

The first line of the `memset` function converts `dest` to a pointer to an `unsigned char` and assigns the resulting value to the `unsigned char` pointer `ptr`. This lets us preserve the value of `dest` so it can be returned in the last line of the function. The remaining two lines of the function form a `while` loop that copies the value of `val` (converted to an `unsigned char`) into each of the first `n` characters of the object pointed to by `dest`. The controlling expression of the `while` loop tests that `n-- > 0`.

The `n` argument is a *loop counter* that's decremented on each iteration of the loop as a side effect of the evaluation of the controlling expression. The loop counter in this case monotonically decreases until a minimum value (0) is reached. The loop performs `n` repetitions, where `n` is less than or equal to the *bound* of the memory referenced by `ptr`.

The pointer `ptr` designates a sequence of objects of type `unsigned char`, from `ptr` through `ptr + n - 1`. The value of `val` is converted to an `unsigned char` and written to each object in turn. If `n` is greater than the bound of the object referenced by `ptr`, the `while` loop will write to memory outside the bounds of this object. This is undefined behavior and a common security flaw, referred to as a *buffer overflow*, or *overrun*. Provided these preconditions are met, the `while` loop will terminate without undefined behavior. In the final iteration of the loop, the controlling expression `n-- > 0` evaluates to 0, causing the loop to terminate.

It is possible to write an *infinite loop*—a loop that runs forever. To avoid writing a `while` loop that inadvertently runs forever, be sure you initialize any objects referenced by the controlling expression before the start of the `while` loop. Also make sure that the controlling expression will change during the `while` loop's execution in a manner that causes the loop to terminate after iterating an appropriate number of times.

The do...while Statement

The `do...while` statement is similar to the `while` statement, except that the evaluation of the controlling expression takes place after each execution of the loop body, rather than before. As a result, the loop body is guaranteed to execute once before the condition is tested. The `do...while` iteration statement has the following syntax:

```
do
  statement
while ( expression );
```

In a `do...while` iteration, *`statement`* is always executed, after which *`expression`* is evaluated. If *`expression`* is not equal to 0, control returns to the top of the loop, and *`statement`* is executed again. Otherwise, execution passes to the statement following the loop.

The `do...while` iteration statement is commonly used in I/O, where it makes sense to read from a stream before testing the state of the stream, as shown in Listing 5-9.

```
#include <stdio.h>
//---snip---
int count; float quant; char units[21], item[21];
do {
  count = fscanf(stdin, "%f%20s of %20s", &quant, units, item);
  fscanf(stdin,"%*[^\n]");
} while (!feof(stdin) && !ferror(stdin));
```

Listing 5-9: Repeatedly accepting a quantity, unit of measure, and an item name from `stdin`

This code inputs a floating-point quantity, a unit of measure (as a string), and an item name (also as a string) from the standard input stream `stdin` until the end-of-file indicator has been set or a read error has occurred. We'll discuss I/O in detail in Chapter 8.

The for Statement

The `for` statement might be the most C-like thing about C. The `for` statement repeatedly executes a statement and is typically used when the number of iterations is known before entering the loop. It has the following syntax:

```
for (clause1; expression2; expression3)
  statement
```

The controlling expression *`expression2`* is evaluated before each execution of the loop body, and *`expression3`* is evaluated after each execution of the loop body. If *`clause1`* is a declaration, the scope of any identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions.

The purpose of *`clause1`*, *`expression2`*, and *`expression3`* is apparent when we translate the `for` statement into an equivalent `while` loop, as shown in Figure 5-1.

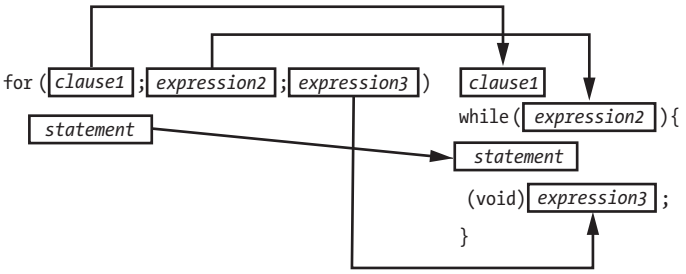

Figure 5-1: Translating a `for` loop into a `while` loop

Listing 5-10 shows a modified version of the `memset` implementation from Listing 5-8; we have replaced the `while` loop with a `for` loop.

```
void *memset(void *dest, int val, size_t n) {
  unsigned char *ptr = (unsigned char *)dest;
  for (size_t i = 0;❶ i < n;❷ ++i❸) {
```

```
    *(ptr + i) = val;
  }
  return dest;
}
```

Listing 5-10: Filling a character array by using a `for` loop

The `for` loop is popular among C programmers because it provides a convenient location for declaring and/or initializing the loop counter ❶, specifying the controlling expression for the loop ❷, and incrementing the loop counter ❸, all concisely on the same line.

The `for` loop can also be somewhat misleading. Let's take the example of a singly linked list in C that declares a `node` structure consisting of a `data` element and a pointer to the `next` node in the list. We also define a pointer `p` to the `node` structure:

```
struct node {
  int data;
  struct node *next;
};
struct node *p;
```

Using the definition of `p`, the following code snippet (used to deallocate the storage for a linked list) erroneously reads the value `p` after it has been freed:

```
for (p = head; p != NULL; p = p->next)
  free(p);
```

Reading `p` after it has been freed is undefined behavior.

If this loop were rewritten as a `while` loop, it would become apparent that the code reads `p` after it was freed:

```
p = head;
while (p != NULL) {
  free(p);
  p = p->next;
}
```

The `for` loop can be confusing because it evaluates *`expression3`* after the main body of the loop even though, lexically, it appears before the loop's body.

The correct way to perform this operation is to save the required pointer before freeing it, like this:

```
for (p = head; p != NULL; p = q) {
  q = p->next;
  free(p);
}
```

You can read more about dynamic memory management in Chapter 6.

Jump Statements

A *jump statement* unconditionally transfers control to another section of the same function when encountered. These are the lowest-level control flow statements and generally correspond closely to the underlying assembly language code.

The goto Statement

Any statement may be preceded by a *label*, which is an identifier followed by a colon. A `goto` statement causes a jump to the statement prefixed by the named label in the enclosing function. The jump is unconditional, meaning it happens every time the `goto` statement is executed. Here is an example of a `goto` statement:

```
  /* executed statements */
  goto location;
  /* skipped statements */
location:
  /* executed statements */
```

Execution continues until the `goto` statement is reached, at which point control jumps to the statement following the label `location`, where execution continues. Statements between the `goto` statement and the label are passed over.

The `goto` statement has had a bad reputation since Edsger Dijkstra wrote a paper in 1968 titled "Go To Statement Considered Harmful" (1968). His criticism was that `goto` statements can result in *spaghetti code* if used haphazardly—code that has a complex and tangled control structure, resulting in a program flow that is conceptually twisted and tangled like a bowl of spaghetti. However, `goto` statements can also make code easier to read if used in a clear, consistent manner.

One helpful way to use `goto` statements is to chain them together to release allocated resources (such as dynamic allocated memory or an open file) when an error occurs and you must leave a function. This scenario occurs when a program allocates multiple resources, each allocation can fail, and resources must be released to prevent leaking. If the first resource allocation fails, no cleanup is needed, because no resources have been allocated. However, if the second resource cannot be allocated, the first resource needs to be released. Similarly, if the third resource cannot be allocated, the second and first resources allocated need to be released, and so forth. This pattern results in duplicated cleanup code, and it can be error-prone because of the duplication and additional complexity.

One solution is to use nested `if` statements, which can also become difficult to read if nested too deeply. Instead, we can use a `goto` chain as shown in Listing 5-11 to release resources.

```
int do_something(void) {
  FILE *file1, *file2;
  object_t *obj;
```

```
  int ret_val = 0; // Initially assume a successful return value

  file1 = fopen("a_file", "w");
  if (file1 == NULL) {
    ret_val = -1;
    goto FAIL_FILE1;
  }

  file2 = fopen("another_file", "w");
  if (file2 == NULL) {
    ret_val = -1;
    goto FAIL_FILE2;
  }

  obj = malloc(sizeof(object_t));
  if (obj == NULL) {
    ret_val = -1;
    goto FAIL_OBJ;
  }

  // Operate on allocated resources

  // Clean up everything
  free(obj);
FAIL_OBJ:  // Otherwise, close only the resources we opened
  fclose(file2);
FAIL_FILE2:
  fclose(file1);
FAIL_FILE1:
  return ret_val;
}
```

Listing 5-11: Using a goto chain to release resources

The code follows a simple pattern: resources are allocated in a certain order, operated upon, and then released in reverse (last in, first out) order. If an error occurs while allocating a resource, the code uses a `goto` to jump to the appropriate location in cleanup code and releases only those resources that have been allocated.

Used in a structured manner such as this, `goto` statements can make code easier to read. A real-world example is the `copy_process` function from `kernel/fork.c` of the Linux kernel, which uses 17 `goto` labels to perform cleanup code when an internal function fails.

The continue Statement

You can use a `continue` statement inside a loop to jump to the end of the loop body, skipping the execution of the remaining statements inside the loop body for the current iteration. For example, the `continue` statement is equivalent to `goto END_LOOP_BODY;` in each of the loops shown in Listing 5-12.

```
while (/* … */) {
  //---snip---
  continue;
  //---snip---
END_LOOP_BODY: ;
}
```

```
do {
  //---snip---
  continue;
  //---snip---
END_LOOP_BODY: ;
} while (/* … */);
```

```
for (/* … */) {
  //---snip---
  continue;
  //---snip---
END_LOOP_BODY: ;
}
```

Listing 5-12: Using the `continue` statement

The `continue` statement is frequently used in conjunction with a conditional statement so that processing may continue with the subsequent loop iteration after the objective of the current loop iteration has been achieved.

The break Statement

A `break` statement terminates execution of a `switch` or iteration statement. We used `break` within a `switch` statement earlier in this chapter. Within a loop, a `break` statement causes the loop to terminate and the program execution to resume at the statement following the loop. For example, the `for` loop in the following example will exit only when the uppercase or lowercase Q key is pressed on the keyboard:

```
#include <stdio.h>
int main(void) {
  char c;
  for(;;) {
    puts("Press any key, Q to quit: ");
    c = toupper(getchar());
    if (c == 'Q') break;
  }
} // Loop exits only when the uppercase or lowercase 'Q' is pressed
```

We typically use `break` statements to discontinue the execution of the loop when the work it was performing has been completed. For example, the `break` statement in Listing 5-13 exits the loop after it finds the specified key in an array. Assuming that `key` is unique in `arr`, the `find_element` function would behave the same without the `break` statement, but could negatively affect performance by executing many more instructions, depending on the length of the array and the point at which `key` is discovered.

```
size_t find_element(size_t len, int arr[len], int key) {
  size_t pos = (size_t)-1;
  // traverse arr and search for key
  for (size_t i = 0; i < len; ++i) {
    if (arr[i] == key) {
      pos = i;
      break;  // terminate loop
    }
  }
```

```
  return pos;
}
```

Listing 5-13: Breaking out of a loop

Because `continue` and `break` bypass part of a loop body, you should use these statements carefully: the code following these statements is not executed.

The return Statement

A `return` statement terminates execution of the current function and returns control to its caller. You've already seen many examples of `return` statements in this book. A function may have 0 or more `return` statements.

A `return` statement can simply return, or it can return an expression. Within a `void` function (a function that doesn't return a value), the `return` statement should simply return. When a function returns a value, the `return` statement should return an expression that produces a value of the return type. If a `return` statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression:

```
int sum(int x, int y, int z) {
  return x + y + z;
}
```

This simple function sums its parameters and returns the sum. The return expression `x + y + z` produces a value of type `int`, which matches the return type of the function. If this expression produced a different type, it would be implicitly converted to an object having the return type of the function. The return expression can also be as simple as returning 0 or 1. The function result may then be used in an expression or assigned to a variable.

Be aware that if control reaches the closing brace of a non-`void` function (a function declared to return a value) without evaluating a `return` statement with an expression, then using the return value of the function call is undefined behavior. For example, the following function fails to return a value when `a` is nonnegative, because the condition `a < 0` is false:

```
int absolute_value(int a) {
  if (a < 0) {
    return -a;
  }
}
```

We can easily repair this defect by providing a return value when `a` is nonnegative, as shown in Listing 5-14.

```
int absolute_value(int a) {
  if (a < 0) {
    return -a;
  }
  return a;
}
```

Listing 5-14: The `absolute_value` function returns a value along all paths.

However, this code still has a bug if two's-complement representation is used (see Chapter 3). Identifying this bug is left as an exercise for you.

Exercises

Try these coding exercises on your own:

1. Modify the function from Listing 5-11 to make it clear to the caller which file could not be opened.
2. Change the `find_element` function from Listing 5-13 to return the position of the key in `a`. Don't forget to return an error indication if the key is not found.
3. Fix the remaining bug in the `absolute_value` function in Listing 5-14.

Summary

In this chapter, you learned about control flow statements:

- Selection statements, such as `if` and `switch`, allow you to select from a set of statements depending on the value of a controlling expression.
- Iteration statements repeatedly execute a loop body until a controlling expression equals 0.
- Jump statements unconditionally transfer control to a new location.

In the next chapter, you'll learn about characters and strings.

6

DYNAMICALLY ALLOCATED MEMORY

In Chapter 2, you learned that every object has a storage duration that determines its lifetime, and that C defines four storage durations: static, thread, automatic, and allocated. In this chapter, you'll learn about *dynamically allocated memory*, which is allocated from the heap at runtime. Dynamically allocated memory is useful when the exact storage requirements for a program are unknown before runtime.

We'll first describe the differences between allocated, static, and automatic storage duration. We'll skip thread storage allocation as this involves parallel execution, which we don't cover here. We'll then explore the functions you can use to allocate and deallocate dynamic memory, common memory allocation errors, and strategies for avoiding them. The terms *memory* and *storage* are used interchangeably in this chapter, in similar fashion to the way they're used in practice.

Storage Duration

Objects occupy *storage*, which might be random-access memory (RAM), read-only memory (ROM), or registers. Storage of allocated duration has significantly different properties than storage of either automatic or static storage duration. First, we'll review automatic and static storage duration—initially described in Chapter 2.

Objects of automatic storage duration are declared within a block or as a function parameter. The lifetime of these objects begins when the block in which they're declared begins execution, and ends when execution of the block ends. If the block is entered recursively, a new object is created each time, each with its own storage.

Objects declared in the file scope have static storage duration. The lifetime of these objects is the entire execution of the program, and their stored value is initialized prior to program startup. You can also declare a variable within a block scope to have static storage duration by using the storage-class specifier `static`.

The Heap and Memory Managers

Dynamically allocated memory has *allocated storage duration*. The lifetime of an allocated object extends from the allocation until the deallocation. Dynamically allocated memory is allocated from the *heap*, which is simply one or more large, subdividable blocks of memory that are managed by the memory manager.

Memory managers are libraries that manage the heap for you by providing implementations of the standard memory management functions described in the following sections. A memory manager runs as part of the client process. The memory manager will request one or more blocks of memory from the operating system and then allocate this memory to the client process when it invokes a memory allocation function.

Memory managers manage unallocated and deallocated memory only. Once memory has been allocated, the caller manages the memory until it's returned. It's the caller's responsibility to ensure that the memory is deallocated, although most implementations will reclaim dynamically allocated memory when the program terminates.

> **MEMORY MANAGER IMPLEMENTATIONS**
>
> Memory managers typically implement a variant of a dynamic storage allocation algorithm described by Donald Knuth (1997). This algorithm uses *boundary tags*—size fields that appear before and after the block of memory returned to the programmer. This size information allows all memory blocks to be traversed from any known block in either direction, allowing the memory manager to coalesce two bordering unused blocks into a larger block to minimize fragmentation.

Fragmentation occurs when memory is allocated and deallocated, resulting in many small blocks of memory but no remaining large blocks. As a result, larger allocations can fail even though the total amount of free memory is sufficient for the allocation. Memory allocated for the client process and memory allocated for internal use within the memory manager is all within the addressable memory space of the client process.

When to Use Dynamically Allocated Memory

Dynamically allocated memory is used when the exact storage requirements for a program are unknown before runtime. Dynamically allocated memory is less efficient than statically allocated memory, because the memory manager needs to find appropriately sized blocks of memory in the runtime heap, and then the caller must explicitly free those blocks when no longer needed, all of which requires additional processing. By default, you should declare objects with either automatic or static storage duration for objects whose sizes are known at compilation time.

Memory leaks occur when dynamically allocated memory that's no longer needed isn't returned to the memory manager. If these memory leaks are severe, the memory manager will eventually be unable to satisfy new requests for storage. Dynamically allocated memory also requires additional processing for housekeeping operations such as *defragmentation* (the consolidation of adjacent free blocks), and the memory manager often uses extra storage for control structures to facilitate these processes.

You would typically use dynamically allocated memory when the size of the storage isn't known at compile time or the number of objects isn't known until runtime. For example, you might use dynamically allocated memory to read a table from a file at runtime, especially if you don't know the number of rows in the table at compile time. Similarly, you might use dynamically allocated memory to create linked lists, hash tables, binary trees, or other data structures for which the number of data elements held in each container is unknown at compile time.

Memory Management Functions

The C Standard Library defines memory management functions for allocating and deallocating dynamic memory. These functions include `malloc`, `aligned_alloc`, `calloc`, and `realloc`. You can call the `free` function to deallocate memory. The OpenBSD `reallocarray` function isn't defined by the C Standard Library but can be useful for memory allocation.

The malloc Function

The `malloc` function allocates space for an object of a specified size whose initial value is indeterminate. In Listing 6-1, we call the `malloc` function to dynamically allocate storage for an object the size of `struct widget`.

```
#include <stdlib.h>
typedef struct {
  char c[10];
  int i;
  double d;
} widget;

❶ widget *p = malloc(sizeof(widget));
❷ if (p == NULL) {
    // Handle allocation error
}
// Continue processing
```

Listing 6-1: Using the `malloc` function to allocate storage for a `widget`

All memory allocation functions accept an argument of type `size_t` that specifies the number of bytes of memory to be allocated ❶. For portability, we use the `sizeof` operator when calculating the size of objects, as the size of objects of various types, such as `int` and `long`, may differ among implementations.

The `malloc` function returns either a null pointer to indicate an error, or a pointer to the allocated space. Therefore, we check whether `malloc` returns a null pointer ❷ and appropriately handle the error.

After the function successfully returns the allocated storage, we can store to members of the `widget` structure through the pointer `p`. For example, `p->i` will access the `int` member of `widget`, while `p->d` will access the `double` member.

Allocating Memory Without Declaring a Type

You can store the return value from `malloc` as a `void` pointer to avoid declaring a type for the referenced object:

```
void *p = malloc(size);
```

Alternately, you can use a `char` pointer, which was the convention before the `void` type was introduced to C:

```
char *p = malloc(size);
```

In either case, the object referenced by `p` has no type until an object is copied into this storage. Once this occurs, the object has the *effective type* of the last object copied into this storage, which imprints the type onto the allocated object. In the following example, the storage referenced by `p` has an effective type of `widget` following the call to `memcpy`.

```
widget w = {"abc", 9, 3.2};
memcpy(p, &w, sizeof(widget));     // coerced to void * pointers
printf("p.i = %d.\n", p->i);
```

Because any type of object can be stored in allocated memory, we can assign the pointers returned by all allocation functions, including `malloc`, to point to any type of object. For example, if an implementation has objects with 1-, 2-, 4-, 8-, and 16-byte alignments, and 16 or more bytes of storage are allocated, a pointer will be returned whose alignment is a multiple of 16.

Casting the Pointer to the Type of the Declared Object

Even expert C programmers disagree about whether to cast the pointer returned by `malloc` to a pointer to the type of the declared object. The following assignment statement casts this pointer as a pointer to `widget`:

```
widget *p = (widget *)malloc(sizeof(widget));
```

Strictly speaking, this cast is unnecessary. C will allow you to implicitly convert a pointer to `void` (the type returned by `malloc`) to a pointer to any type of object for which the resulting pointer is correctly aligned. (Otherwise, the behavior is undefined.) Casting the result of `malloc` to the intended pointer type enables the compiler to catch inadvertent pointer conversions, as well as differences between the size of the allocation and the size of the pointed-to type in the cast expression.

Examples in this book typically use a cast, but either style is acceptable. See CERT C rule MEM02-C (Immediately cast the result of a memory allocation function call into a pointer to the allocated type) for more on this topic.

Reading Uninitialized Memory

The contents of memory returned from `malloc` are *uninitialized*, which means it contains indeterminate values. Reading uninitialized memory is never a good idea, and you should think of it as undefined behavior—if you'd like to know more, I wrote an in-depth article on *uninitialized reads* (Seacord 2017). The `malloc` function does not initialize the returned memory, because the expectation is that you will overwrite this memory anyway.

Even so, beginners commonly make the mistake of assuming that the memory returned by `malloc` contains zeros. The program shown in Listing 6-2 makes this exact error.

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

int main(void) {
  char *str = (char *)malloc(16);
  if (str) {
    strncpy(str, "123456789abcdef", 15);
```

```
    printf("str = %s.\n", str);
    free(str);
    return EXIT_SUCCESS;
  }
  return EXIT_FAILURE;
}
```

Listing 6-2: Initialization error

This program allocates 16 bytes of memory by calling `malloc`, and then uses `strncpy` to copy the first 15 bytes of a string into the allocated memory. The programmer attempts to create a properly null-terminated string by copying one less byte than the size of the allocated memory. In doing so, the programmer assumes that the allocated storage already contains a 0 value to serve as the null byte. However, the storage could easily contain nonzero values, in which case the string wouldn't be properly null-terminated, and the call to `printf` would result in undefined behavior.

A common solution is to write a null character into the last byte of the allocated storage, as follows:

```
strncpy(str, "123456789abcdef", 15);
str[15] = '\0';
```

If the source string is less than 15 bytes, the null termination character will be copied, and the assignment at ❶ is unnecessary. If the source string is 15 bytes or longer, adding this assignment will ensure that the string is properly null-terminated.

The aligned_alloc Function

The `aligned_alloc` function is similar to the `malloc` function, except that it requires you to define an alignment as well as a size for the allocated object. The function has the following signature, where `size` specifies the object's size, and `alignment` specifies its alignment:

```
void *aligned_alloc(size_t alignment, size_t size);
```

C11 introduced the `aligned_alloc` function because some hardware has stricter-than-normal memory alignment requirements. Although C requires the dynamically allocated memory from `malloc` to be sufficiently aligned for all standard types, including arrays and structures, you might occasionally need to override the compiler's default choices.

Typically, we use the `aligned_alloc` function to request stricter alignment than the default (in other words, a larger power of two). If the value of `alignment` is not a valid alignment supported by the implementation, the function fails by returning a null pointer. See Chapter 2 for more information on alignment.

The calloc Function

The `calloc` function allocates storage for an array of `nmemb` objects, each of whose size is `size` bytes. It has the following signature:

```
void *calloc(size_t nmemb, size_t size);
```

This function initializes the storage to all zero-valued bytes. This zero value might not be the same one used to represent floating-point zero or null-pointer constants. You can also use the `calloc` function to allocate storage for a single object, which can be thought of as an array of one element.

Internally, the `calloc` function works by multiplying `nmemb` by `size` to determine the required number of bytes to allocate. Historically, some `calloc` implementations failed to validate that these values wouldn't overflow when multiplied. Modern implementations of `calloc` perform this overflow check and return a null pointer if the product cannot be represented in `size_t`.

The realloc Function

The `realloc` function increases or decreases the size of previously allocated storage. It takes a pointer to some memory allocated by an earlier call to `aligned_alloc`, `malloc`, `calloc`, or `realloc` (or a null pointer) and a size, and has the following signature:

```
void *realloc(void *ptr, size_t size);
```

You can use the `realloc` function to grow or (less commonly) shrink the size of an array.

Avoiding Memory Leaks

To avoid introducing bugs when you use `realloc`, you should understand how the function is (conceptually) implemented. The `realloc` function typically calls the `malloc` function to allocate new storage, and then copies the contents of the old storage to the new storage up to the minimum of the old and new sizes. If the newly allocated storage is larger than the old contents, `realloc` leaves the additional storage uninitialized. If `realloc` succeeds in allocating the new object, it calls `free` to deallocate the old object. If the allocation fails, the `realloc` function retains the old object data at the same address and returns a null pointer. A call to `realloc` can fail, for example, when insufficient memory is available to allocate the requested number of bytes. The following use of `realloc` contains an error:

```
size += 50;
if ((p = realloc(p, size)) == NULL) return NULL;
```

In this example, `size` is incremented by 50 before calling `realloc` to increase the size of the storage referenced by `p`. If the call to `realloc` fails, `p` is assigned the value `NULL`, but `realloc` doesn't deallocate the storage referenced by `p`, resulting in this memory being leaked.

Listing 6-3 demonstrates the correct use of the `realloc` function.

```
void *p2;
void *p = malloc(100);
//---snip---
if ((nsize == 0) || (p2 = realloc(p, nsize)) == NULL) {
  free(p);
  return NULL;
}
p = p2;
```

Listing 6-3: Correct use of the `realloc` function

This code snippet declares two variables, `p` and `p2`. The variable `p` refers to the dynamically allocated memory returned by `malloc`, and `p2` starts out uninitialized. Eventually, this memory is resized, which we accomplish by calling the `realloc` function with the pointer `p` and the new size `nsize`. The return value from `realloc` is assigned to `p2` to avoid overwriting the pointer stored in `p`. If `realloc` returns a null pointer, the memory referenced by `p` is freed, and the function returns a null pointer. If `realloc` succeeds and returns a pointer to an `nsize` allocation, `p` is assigned the pointer to the newly reallocated storage, and execution continues.

This code also includes a test for a zero-byte allocation. You should avoid passing the `realloc` function a value of 0 as the size argument, as this is effectively undefined behavior (and is actually undefined behavior in C2x).

If the following call to the `realloc` function does not return a null pointer, the address stored in `p` is invalid and can no longer be read:

```
newp = realloc(p, ...);
```

In particular, the following test is not allowed:

```
if (newp != p) {
  // update pointers to reallocated memory
}
```

Any pointers that reference the memory previously pointed to by `p` must be updated to reference the memory pointed to by `newp` after the call to `realloc` regardless of whether `realloc` kept the same address for the storage.

One solution to this problem is to go through an extra indirection, sometimes called a *handle*. If all uses of the reallocated pointer are indirect, they will all be updated when that pointer is reassigned.

Calling realloc with a Null Pointer

Calling `realloc` with a null pointer is equivalent to calling `malloc`. Provided `newsize` isn't equal to 0, the following code

```
if (p == NULL)
  newp = malloc(newsize);
else
  newp = realloc(p, newsize);
```

can be replaced with

```
newp = realloc(p, newsize);
```

The first, longer version of this code calls `malloc` the first time storage is allocated, and `realloc` to adjust the size later as required. Because calling `realloc` with a null pointer is equivalent to calling `malloc`, the second version concisely accomplishes the same thing.

The reallocarray Function

The OpenBSD `reallocarray` function can reallocate storage for an array, but also provides overflow checking for array size calculations. This saves you from having to perform these checks. The `reallocarray` function has the following signature:

```
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

The `reallocarray` function allocates storage for `nmemb` members of size `size` and checks for integer overflow in the calculation `nmemb * size`. Other platforms, including the GNU C Library (libc), have adopted this function, and it has been proposed for inclusion in the next revision of the POSIX standard. The `reallocarray` function doesn't zero out the allocated storage.

As we have seen in previous chapters, integer overflow is a serious problem that can result in buffer overflows and other security vulnerabilities. In the following code, for example, the expression `num * size` might overflow before being passed as the size argument in the following call to `realloc`:

```
if ((newp = realloc(p, num * size)) == NULL) {
  //---snip---
```

The `reallocarray` function is useful when two values are multiplied to determine the size of the allocation:

```
if ((newp = reallocarray(p, num, size)) == NULL) {
  //---snip---
```

This call to the `reallocarray` function will fail and return a null pointer if `num * size` would otherwise overflow.

The free Function

You should deallocate dynamically allocated memory when it's no longer needed by calling the `free` function. Deallocating memory is important because it allows that memory to be reused, reducing the chances that you'll exhaust the available memory, and often providing more efficient use of the heap.

We deallocate memory by passing a pointer to that memory to the `free` function, which has the following signature:

```
void free(void *ptr);
```

The `ptr` value must have been returned by a previous call to `aligned_alloc`, `malloc`, `calloc`, or `realloc`. CERT C rule MEM34-C (Only free memory allocated dynamically) discusses what happens when the value is not returned. Memory is a limited resource and so must be reclaimed.

If we call `free` with a null-pointer argument, nothing happens, and the `free` function simply returns:

```
  char *ptr = NULL;
  free(ptr);
```

Avoiding Double-Free Vulnerabilities

If you call the `free` function on the same pointer more than once, undefined behavior occurs. These defects can result in a security flaw known as a *double-free vulnerability*. One possible consequence is that they can be exploited to execute arbitrary code with the permissions of the vulnerable process. The full effects of double-free vulnerabilities are beyond the scope of this book, but I discuss them in detail in *Secure Coding in C and C++* (Seacord 2013). Double-free vulnerabilities are especially common in error-handling code, as programmers attempt to free allocated resources.

Another common error is to access memory that has already been freed. This type of error frequently goes undiagnosed, because the code might appear to work, but fails in unexpected ways away from the actual error. In Listing 6-4, taken from an actual application, the argument to `close` is invalid, because the storage formerly pointed to by `dirp` has been reclaimed by the second call to `free`:

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

int closedir(DIR *dirp) {
  free(dirp->d_buf);
  free(dirp);
  return close(dirp->d_fd);  // dirp has already been freed
}
```

Listing 6-4: Accessing already freed memory

We refer to pointers to already freed memory as *dangling pointers*. Dangling pointers are a potential source of errors (like a banana peel on the floor) because they can be used to write to memory that has already been freed or passed to the `free` function, resulting in double-free vulnerabilities. See CERT C rule MEM30-C (Do not access freed memory) for more information on these topics.

Setting the Pointer to Null

To limit the opportunity for defects involving dangling pointers, set the pointer to `NULL` after completing a call to `free`:

```
char *ptr = malloc(16);
//---snip---
free(ptr);
ptr = NULL;
```

Any future attempt to dereference the pointer will usually result in a crash (increasing the likelihood that the error is detected during implementation and testing). If the pointer is set to `NULL`, the memory can be freed multiple times without consequence. Unfortunately, the `free` function cannot set the pointer to `NULL` itself, because it's passed a copy of the pointer, not the actual pointer.

Memory States

Dynamically allocated memory can exist in one of three states shown in Figure 6-1: unallocated and uninitialized within the memory manager, allocated but uninitialized, and allocated and initialized. Calls to the `malloc` and `free` functions, as well as writing the memory, cause the memory to transition from one state to another.

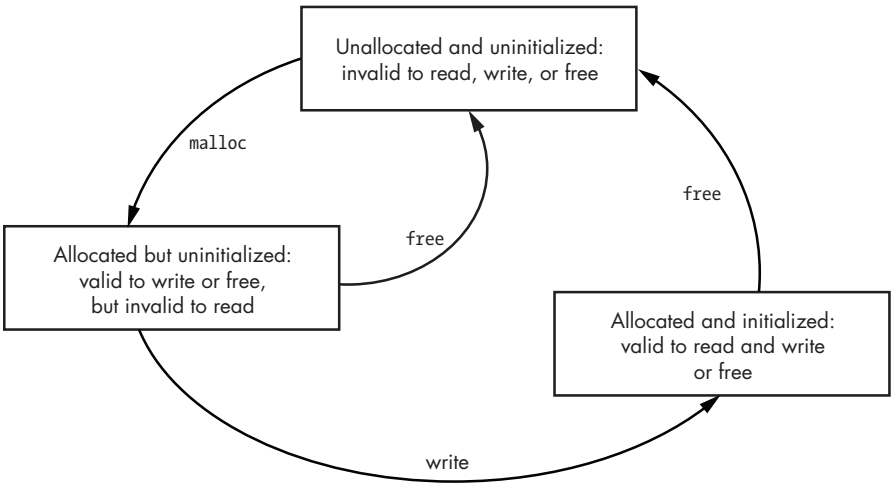

Figure 6-1: Memory states

Different operations are valid depending on the state of the memory. Avoid any operation on memory that is not shown as valid or is explicitly listed as invalid. This applies to each byte of memory, as bytes that have been initialized can be read, but bytes that have not been initialized must not be read.

Flexible Array Members

Allocating storage for a structure that contains an array has always been a little tricky in C. There's no problem if the array has a fixed number of elements, as the size of the structure can easily be determined. Developers, however, frequently need to declare an array with a flexible size, and previously, C offered no easy way to do this.

Flexible array members let you declare and allocate storage for a structure with any number of fixed members, where the last member is an array of unknown size. Starting with C99, the last member of a `struct` with more than one member can have *incomplete array type*, which means that the array has an unknown size. This enables you to wait until runtime to specify the array size. A flexible array member allows you to access a variable-length object.

For example, Listing 6-5 shows the use of a flexible array member `data` in `widget`. We dynamically allocate storage for the object by calling the `malloc` function.

```
#include <stdlib.h>

typedef struct {
  size_t num;
❶ int data[];
} widget;

void *func(size_t array_size) {
❷ widget *p = (widget *)malloc(sizeof(widget) + sizeof(int) * array_size);
  if (p == NULL) {
    return NULL;
  }

  p->num = array_size;
  for (size_t i = 0; i < p->num; ++i) {
  ❸ p->data[i] = 17;
  }
}
```

Listing 6-5: Flexible array members

We first declare a `struct` whose last member, the `data` array ❶, is an incomplete type (with no specified size). We then allocate storage for the entire `struct` ❷. When computing the size of a `struct` containing a flexible array member using the `sizeof` operator, the flexible array member is ignored. Therefore, we must explicitly include an appropriate size for the flexible array member when allocating storage. To accomplish this, we allocate additional bytes for the array by multiplying the number of elements

in the array (`array_size`) by the size of each element (`sizeof(int)`). This program assumes that the value of `array_size` is such that when multiplied by `sizeof(int)` an overflow will not occur.

We can access this storage by using a `.` or `->` operator ❸, as if the storage had been allocated as `data[array_size]`. See CERT C rule MEM33-C (Allocate and copy structures containing a flexible array member dynamically) for more information on allocating and copying structures containing flexible array members.

Prior to C99, multiple compilers supported a similar "`struct` hack" using a variety of syntaxes. CERT C rule DCL38-C (Use the correct syntax when declaring a flexible array member) is a reminder to use the syntax specified in C99 and later versions of the C Standard.

Other Dynamically Allocated Storage

Beyond the memory management functions that allow you to allocate memory from the heap, C has language and library features that allow storage to be dynamically allocated. This storage is typically allocated in the stack frame of the caller (the C Standard does not define a stack, but it is a common implementation feature). A *stack* is a last-in-first-out (LIFO) data structure that supports nested invocation of functions at runtime. Each function invocation creates a *stack frame* in which local variables (of automatic storage duration) and other data specific to that invocation of the function can be stored.

The alloca Function

For performance reasons, the `alloca` function, supported by some implementations, allows dynamic allocation at runtime from the stack rather than the heap. This memory is automatically released when the function that called `alloca` returns. The `alloca` function is an *intrinsic* (or *built-in*) function, which means that its implementation is handled specially by the compiler. This allows the compiler to substitute a sequence of automatically generated instructions for the original function call. For example, on the x86 architecture, the compiler substitutes a call to `alloca` with a single instruction to adjust the stack pointer to accommodate the additional storage.

The `alloca` function originated in an early version of the Unix operating system from Bell Laboratories but is not defined by the C Standard Library or POSIX. Listing 6-6 shows an example function called `printerr` that uses the `alloca` function to allocate storage for an error string before printing it out to `stderr`.

```
void printerr(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum) + 1;
  char *msg = (char *)alloca(size);
  if (strerror_s(msg, size, errnum) != 0) {
    fputs(msg, stderr);
  }
  else {
```

```
    fputs("unknown error", stderr);
  }
}
```

Listing 6-6: The `printerr` function

The `printerr` function takes a single argument, `errnum`, of type `errno_t`. In the first line of the function, we call the `strerrorlen_s` function to determine the length of the error string associated with this particular error number. Once we know the size of the array that we need to allocate to hold the error string, we can call the `alloca` function to efficiently allocate storage for the array. We then retrieve the error string by calling the `strerror_s` function and store the result in the newly allocated storage referenced by `msg`. Assuming the `strerror_s` function succeeds, we output the error message; otherwise, we output `unknown error`. This `printerr` function is written to demonstrate the use of `alloca` and is more complicated than it actually needs to be.

The `alloca` function can be tricky to use. First, the call to `alloca` can make allocations that exceed the bounds of the stack. However, the `alloca` function does not return a null pointer value, so there is no way to check for the error. For this reason, it's critically important to avoid using `alloca` with large or unbounded allocations. The call to `strerrorlen_s` in this example should return a reasonable allocation size.

A further problem with the `alloca` function is that programmers may become confused by having to free calls to `malloc` but not to `alloca`. Calling `free` on a pointer not obtained by calling `aligned_alloc`, `calloc`, `realloc`, or `malloc` is a serious error. Compilers also tend to avoid inlining functions that call `alloca`. For these reasons, the use of `alloca` is discouraged.

The GCC compiler provides a `-Walloca` flag that diagnoses all calls to the `alloca` function, and a `-Walloca-larger-than=`*`size`* flag that diagnoses any call to the `alloca` function when the requested memory is more than *`size`*.

Variable-Length Arrays

Variable-length arrays (VLAs), introduced in C99, are arrays that you can declare by using a variable to specify the array dimensions, allowing you to specify their size at runtime. The size of the array cannot be modified after you create it. VLAs are useful when you don't know the number of elements in the array until runtime. All VLA declarations must be at either block scope or function prototype scope. The next sections show examples of each.

Block Scope

The following function `func` declares the variable-length array `vla` of size `size` as an automatic variable with *block scope*.

```
void func(size_t size) {
  int vla[size];
  //---snip---
}
```

The array is allocated in the stack frame and freed when the current frame exits—similar to the `alloca` function. Listing 6-7 replaces the call to `alloca` in the `printerr` function from Listing 6-6 with a VLA. The change modifies just a single line of code (shown in bold).

```
void print_error(int errnum) {
  size_t size = strerrorlen_s(errnum) + 1;
  char msg[size];
  if (strerror_s(msg, size, errnum) != 0) {
    fputs(msg, stderr);
  }
  else {
    fputs("unknown error", stderr);
  }
}
```

Listing 6-7: The `print_error` function rewritten to use a VLA

The main advantage of using VLAs instead of the `alloca` function is that the syntax matches the programmer's model of how arrays with automatic storage duration work—that is, there's no need to explicitly free the storage.

VLAs share some of the problems of the `alloca` function, in that they can attempt to make allocations that exceed the bounds of the stack. Unfortunately, there's no portable way to determine the remaining stack space to detect such an error. Also, the calculation of the array's size could overflow when the size you provide is multiplied by the size of each element. For these reasons, it's important to validate the size of the array before declaring it to avoid overly large or incorrectly sized allocations. This can be especially important in functions that are called recursively, because a complete new set of automatic variables for the functions (including these arrays) will be created for each recursion.

You should determine whether you have sufficient stack space in the worst-case scenario (maximum-sized allocations with deep recursions). On some implementations, it's also possible to pass a negative size to the VLA, so make sure your size is represented as a `size_t` or other unsigned type. See CERT C rule ARR32-C (Ensure size arguments for variable-length arrays are in a valid range) for more information. For GCC, you can use the `-Wvla-larger-than=size` flag to diagnose definitions of VLAs that either exceed the specified size or whose bound is not sufficiently constrained.

Finally, another interesting and possibly unexpected behavior occurs when calling `sizeof` on a VLA. The compiler usually performs the `sizeof` operation at compile time. However, if the expression changes the size of the array, it will be evaluated at runtime, including any side effects. The same is true of `typedef`, as shown by the program in Listing 6-8.

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

int main(void) {
```

```
  size_t size = 12;
  printf("%zu\n", size); // prints 12
  (void)sizeof(int[size++]);
  printf("%zu\n", size); // prints 13
  typedef int foo[size++];
  printf("%zu\n", size); // prints 14
}
```

Listing 6-8: Unexpected side effects

In this simple test program, we declare a variable `size` of type `size_t` and initialize it to 12. We then call the `sizeof` operator with `int[size++]` as the argument. Because this expression changes the size of the array, `size` is incremented and is now equal to 13. The `typedef` similarly increments the value of `size` to 14.

Function Prototype Scope

You can also declare VLAs as function parameters. Remember from Chapter 2 that, when used in an expression, an array is converted to a pointer to the first element of the array. This means that we must add an explicit parameter to specify the size of the array—for example, the `n` parameter in the signature for `memset`:

```
void *memset(void *s, int c, size_t n);
```

When you call such a function, `n` should accurately represent the size of the array referenced by `s`. Undefined behavior results if this size is larger than the array.

When declaring a function to take a VLA as an argument that specifies a size, we must declare the size of the array before referencing the size in the array declaration. We could, for example, modify the signature for the `memset` function as follows to take a VLA:

```
void *memset_vla(size_t n, char s[n], int c);
```

Here, we've changed the order of the parameters so that the variable `n` of type `size_t` is declared before we use it in the array declaration. The array argument `s` is still demoted to a pointer, and no storage is allocated as a result of this declaration. When calling this function, you are responsible for declaring the actual storage for the array referenced by `s` and ensuring that `n` is a valid size for it.

VLAs can generalize your functions, making them more useful. For example, the `matrix_sum` function sums all the values in a two-dimensional array. The following version of this function accepts a matrix with a fixed column size:

```
int matrix_sum(size_t rows, int m[][4]);
```

When passing a multidimensional array to a function, the size information for the initial dimension of the array is lost, but it still needs to be passed in as an argument. This information is provided by the `rows` parameter in this example. You can call this function to sum the values of any matrix with exactly four columns, as shown in Listing 6-9.

```
int main(void) {
  int m1[5][4];
  int m2[100][4];
  int m3[2][4];
  printf("%d.\n", matrix_sum(5, m1));
  printf("%d.\n",  matrix_sum(100, m2));
  printf("%d.\n", matrix_sum(2, m3));
}
```

Listing 6-9: Summing matrices with four columns

This is all well and good until you need to sum the values of a matrix that doesn't have four columns. For example, changing `m3` to have five columns would result in a warning such as this:

```
warning: incompatible pointer types passing 'int [2][5]' to parameter of type 'int (*)[4]'
```

To handle this case, you'd have to write a new function with a signature that matches the new dimensions of the multidimensional array. The problem with this approach, then, is that it fails to generalize sufficiently.

Instead of doing that, we can rewrite the `matrix_sum` function to use a VLA, as shown in Listing 6-10. This change allows us to call `matrix_sum` with matrices of any dimension.

```
int matrix_sum(size_t rows, size_t cols, int m[rows][cols]) {
  int total = 0;

  for (size_t r = 0; r < rows; r++)
    for (size_t c = 0; c < cols; c++)
      total += m[r][c];
  return total;
}
```

Listing 6-10: Using a VLA as a function parameter

Again, no storage is allocated by either the function declaration or the function definition. You need to allocate the storage for the matrix separately, and its dimensions must match those passed to the function as the `rows` and `cols` arguments. Failing to do so can result in undefined behavior.

Debugging Allocated Storage Problems

As noted earlier in this chapter, improper memory management can lead to errors like leaking memory, reading from or writing to freed memory, and freeing memory more than once. One way to avoid some of these problems

is to set pointers to NULL after calling free, as we've already discussed. Another strategy is to keep your dynamic memory management as simple as possible. For example, you should allocate and free memory in the same module, at the same level of abstraction, rather than freeing memory in subroutines, which leads to confusion about if, when, and where memory was freed.

A third option is to use *dynamic analysis tools*, which detect and report memory errors. These tools, as well as general approaches to debugging, testing, and analysis, are discussed in Chapter 11. In this section, we'll cover one of these tools: dmalloc.

Dmalloc

The *debug memory allocation* (dmalloc) library created by Gray Watson replaces malloc, realloc, calloc, free, and other memory management features with routines that provide debugging facilities that you can configure at runtime. The library has been tested on a variety of platforms.

Follow the installation directions provided at *https://dmalloc.com/* to configure, build, and install the library. Listing 6-11 shows you how to enable dmalloc to report the file and line numbers of calls that generate problems. This listing contains a short program that prints out some usage information and exits (it would typically be part of a longer program). Include the lines shown in bold font to allow dmalloc to report the file and line numbers of calls that cause problems.

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

#ifdef DMALLOC
#include "dmalloc.h"
#endif

void usage(char *msg) {
  fprintf(stderr, "%s", msg);
  free(msg);
  return;
}

int main(int argc, char *argv[]) {
  if (argc != 3 && argc != 4) {
    /* the error message won't be more than 80 chars */
    char *errmsg = (char *)malloc(80);
    sprintf(
      errmsg,
      "Sorry %s,\nUsage: caesar secret_file keys_file [output_file]\n",
      getenv("USER")
    );
    usage(errmsg);
    free(errmsg);
    exit(EXIT_FAILURE);
  }
```

```
  //---snip---

  exit(EXIT_SUCCESS);
}
```

Listing 6-11: Catching a memory bug with `dmalloc`

I'll show the output in a moment, but we need to discuss a few things first. The `dmalloc` distribution also comes with a command line utility. You can get further information on how to use this utility by running the following:

```
% dmalloc --usage
```

Before debugging your program with `dmalloc`, enter the following at the command line:

```
% dmalloc -l logfile -i 100 low
```

This command will set the logfile name to *logfile*, and instruct the library to perform a check every 100 invocations, specified in the `-i` argument. If you specify a larger number as the `-i` argument, `dmalloc` will check the heap less often, and your code will run faster; lower numbers are more likely to catch memory problems. The third command line argument enables a `low` number of debug features. Other options include `runtime` for minimal checking, or `medium` or `high` for more extensive heap verification.

After executing this command, we can compile the program by using GCC as follows:

```
% gcc -DDMALLOC caesar.c -ocaesar -ldmalloc
```

When you run the program, you should see the following error:

```
% ./caesar
Sorry student,
Usage: caesar secret_file keys_file [output_file]
debug-malloc library: dumping program, fatal error
  Error: tried to free previously freed pointer (err 61)
Aborted (core dumped)
```

And if you examine the logfile, you'll find the following information:

```
% more logfile
1571549757: 3: Dmalloc version '5.5.2' from 'https://dmalloc.com/'
1571549757: 3: flags = 0x4e48503, logfile 'logfile'
1571549757: 3: interval = 100, addr = 0, seen # = 0, limit = 0
1571549757: 3: starting time = 1571549757
1571549757: 3: process pid = 29531
1571549757: 3:   error details: finding address in heap
1571549757: 3:   pointer '0x7ff010812f88' from 'caesar.c:29' prev access 'unknown'
1571549757: 3: ERROR: free: tried to free previously freed pointer (err 61)
```

These messages indicate that we've attempted to free the storage referenced by `errmsg` twice, first in the `usage` function and then in `main`, which constitutes a double-free vulnerability. Of course, this is just a single example of the types of bugs that `dmalloc` can detect, and other defects exist in the simple program we are testing. We'll discuss other dynamic analysis tools in Chapter 11, along with further suggestions for how to use them.

Safety-Critical Systems

Systems with high safety requirements frequently ban the use of dynamic memory, because memory managers can have unpredictable behavior that significantly impacts performance. Forcing all applications to live within a fixed, pre-allocated area of memory can eliminate many of these problems and make it easier to verify memory use. In the absence of recursion, `alloca`, and VLAs (also prohibited in safety-critical systems), an upper bound on the use of stack memory can be derived statically, making it possible to prove that sufficient storage exists to execute the functionality of the application for all possible inputs.

GCC also has a `-Wvla` flag that warns if a VLA is used, and a `-Wvla-larger-than=`*`byte-size`* flag that warns for declarations of VLAs whose size is either unbounded, or bounded by an argument that allows the array size to exceed *`byte-size`* bytes.

Exercises

Try these code exercises on your own:

1. Repair the use-after-free defect from Listing 6-4.
2. Perform additional testing of the program from Listing 6-11 by using `dmalloc`. Try varying inputs to the program to identify other memory management defects.

Summary

In this chapter, you learned about working with memory that has allocated storage duration, and how this differs from objects of either automatic or static storage duration. We described the heap and memory managers and each of the standard memory management functions. We identified some common causes of errors when using dynamic memory, such as leaks, double-free vulnerabilities, and some mitigations to help avoid these problems.

We also covered some more specialized memory allocation topics such as flexible array members, the `alloca` function, and variable-length arrays. We concluded the chapter with a discussion of debugging allocated storage problems by using the `dmalloc` library.

In the next chapter, you'll learn about characters and strings.

7

CHARACTERS AND STRINGS

Strings are such an important and useful data type that nearly every programming language implements them in some form. Often used to represent text, strings constitute most of the data exchanged between an end user and a program, including text input fields, command line arguments, environment variables, and console input.

In C, the string data type is modeled on the idea of a formal string (Hopcroft 1979):

> Let Σ be a non-empty finite set of characters, called the alphabet. A string over Σ is any finite sequence of characters from Σ. For example, if Σ = {0, 1}, then 01011 is a string over Σ.

In this chapter, we'll talk about the various character sets, including ASCII and Unicode, that can be used to compose strings (the *alphabet* from

the formal definition). We'll cover how strings are represented and manipulated using the legacy functions from the C Standard Library, the bounds-checked interfaces, and POSIX and Windows APIs.

Characters

The characters that people use to communicate aren't naturally understood by digital systems, which operate on bits. To process characters, digital systems use *character encodings* that assign unique integer values, referred to as *code points*, to designate specific characters. As you'll see, there are multiple ways to encode the same notional character in your program. Common standards used by C implementations for encoding characters include Unicode, ASCII, Extended ASCII, ISO 8859-1, Shift-JIS, and EBCDIC.[9]

ASCII

The 7-bit American Standard Code for Information Interchange, better known as *7-bit ASCII*, specifies a set of 128 characters and their coded representation (ANSI X3.4-1986). Characters up through `0x1f` are control characters, such as null, backspace, and horizontal tab. Characters from `0x20` through `0x7e` are all printable characters such as letters, digits, and symbols.

We often refer to this standard with the updated name *US-ASCII* to clarify that this system was developed in the United States and focuses on the typographical symbols predominantly used in this country. Most modern character encoding schemes are based on US-ASCII, although they support many additional characters.

Characters in the `0x80-0xFF` range are not defined by US-ASCII but are part of the 8-bit character encoding known as *Extended ASCII*. Numerous encodings for these ranges exist, and the actual mapping depends on the code page. A *code page* is a character encoding that maps a set of printable characters and control characters to unique numbers.

Unicode

Unicode has become the universal character encoding standard for representing text in computer processing. It supports a much wider range of characters than ASCII does; the current Unicode Standard (Unicode 2020) encodes characters in the range `U+0000` to `U+10FFFF`, which amounts to a 21-bit code space. An individual Unicode value is expressed as `U+` followed by four or more hexadecimal digits in printed text. The Unicode characters `U+0000` to `U+007F` are identical to those in US-ASCII, and the range `U+0000` to `U+00FF` is identical to ISO 8859-1 (Latin-1), consisting of characters from the Latin script used throughout the Americas, Western Europe, Oceania, and much of Africa.

9. Unicode and ASCII are explicitly referenced by the C Standard.

Unicode organizes code points into *planes*, which are continuous groups of 65,536 code points. There are 17 planes, identified by the numbers 0 to 16. The most commonly used characters, including those found in major, older encoding standards, have been placed into the first plane (`0x0000` to `0xFFFF`), which is called the *basic multilingual plane (BMP),* or Plane 0.

Unicode also specifies several *Unicode transformation formats (UTFs).* UTFs are character encoding formats that assign each Unicode scalar value to a unique code unit sequence. A *Unicode scalar value* is any Unicode code point except high-surrogate and low-surrogate code points. A *code unit* is the minimal bit combination that can represent encoded text for processing or interchange. The Unicode Standard defines three UTFs to allow for code units of various sizes:

UTF-8 Represents each character as a sequence of one to four 8-bit code units

UTF-16 Represents each character as a sequence of one or two 16-bit code units

UTF-32 Represents each character as a single 32-bit code unit

The UTF-8 encoding is the dominant encoding for POSIX operating systems. It has the following desirable properties:

- It encodes US-ASCII characters (`U+0000` to `U+007F`) as single bytes in the range `0x00` to `0x7F`. This means that files and strings that contain only 7-bit ASCII characters have the same encoding under both ASCII and UTF-8.
- Using a null byte to terminate a string (a topic we'll discuss later) works the same as for an ASCII string.
- All currently defined Unicode code points can be encoded using between 1 and 4 bytes.
- Unicode is designed to make it easy to identify character boundaries by scanning for well-defined bit patterns in either direction.

On Windows, you can compile and link your programs with the Visual C++ `/utf8` flag to set the source and execution character sets as UTF-8. You'll also need to configure Windows to use Unicode UTF-8 for worldwide language support, unless this changes in the future.

UTF-16 is currently the dominant encoding for Windows operating systems. Like UTF-8, UTF-16 is a variable-width encoding. As just mentioned, the BMP consists of characters from `U+0000` to `U+FFFF`. Characters whose code points are greater than `U+FFFF` are called *supplementary characters.* Supplementary characters are defined by a pair of code units called *surrogates.* The first code unit is from the high-surrogates range (`U+D800-U+DBFF`), and the second code unit is from the low-surrogates range (`U+DC00-U+DFFF`).

UTF-32 is a fixed-length encoding, unlike other Unicode transformation formats, which are variable-length encodings. The main advantage of UTF-32 is that the Unicode code points are directly indexed, meaning that

you can find the *n*th code point in a sequence of code points in constant O(1) time. In contrast, a variable-length encoding requires accessing each code point sequentially to find the *n*th code point in a sequence.

Source and Execution Character Sets

No universally accepted character encoding existed when C was originally standardized, so it was designed to work with a wide variety of character representations. Instead of specifying a character encoding like Java, each C implementation defines both a *source character set* in which source files are written and an *execution character set* used for character and string literals at compile time.

Both the source and execution character sets must contain encodings for the uppercase and lowercase letters of the Latin alphabet; the 10 decimal digits; the 29 graphic characters; and the space, horizontal tab, vertical tab, form feed, and newline characters. The execution character set also includes alert, backspace, carriage return, and null characters.

The character conversion and classification functions (such as `isdigit`) are evaluated at runtime, based on the locale-determined encoding in effect at the time of the call. *Locales* define local conventions of nationality, culture, and language.

Data Types

C defines several data types to represent character data, some of which we have already seen. In particular, C offers the unadorned `char` type to represent *narrow characters* (those that can be represented in as few as 8 bits), and the `wchar_t` type to represent *wide characters* (those that may require more than 8 bits).

char

As we have already seen, `char` is an integer type, but each implementation defines whether it is signed or unsigned—meaning that in portable code, you cannot assume either.

Use the `char` type for character data (where signedness has no meaning) and not for integer data (where signedness is important). The `char` type can be safely used to represent 7-bit character encodings, such as US-ASCII. For these encodings, the high-order bits are always 0, so you don't have to be concerned about sign extension when a value of type `char` is converted to `int` and implementation defined as a signed type.

The `char` type can also be used to represent 8-bit character encodings such as Extended ASCII, ISO/IEC 8859, EBCDIC, and UTF-8. These 8-bit character encodings can be problematic on implementations that define `char` as an 8-bit signed type. For example, the following code prints the string `end of file` when an `EOF` is detected:

```
char c = 'ÿ';  // extended character
if (c == EOF) puts("end of file");
```

Assuming the implementation-defined execution character set is ISO/IEC 8859-1, the Latin small letter y with diaeresis (ÿ) is defined to have the representation `255` (`0xFF`). For implementations in which `char` is defined as a signed type, `c` will be sign-extended to the width of `signed int`, making the ÿ character indistinguishable from `EOF` because they now have the same representation.

A similar problem occurs when using the character classification functions defined in `<ctype.h>`. These library functions accept a character argument as an `int` or the value of the macro `EOF`, and return `true` if the value belongs to a set of characters defined by the description of the function. For example, the `isdigit` function tests whether the character is a decimal-digit character in the current locale. Any argument value that isn't a valid character or `EOF` will result in undefined behavior.

To avoid undefined behavior when invoking these functions, cast `c` to `unsigned char` before the integer promotions, as shown here:

```
char c = 'ÿ';
if (isdigit((unsigned char)c)) {
  puts("c is a digit");
}
```

The value stored in `c` is zero-extended to the width of `signed int`, eliminating the undefined behavior because the resulting value can still be represented as an `unsigned char`.

int

Use the `int` type for data that could be either `EOF` (a negative value) or character data interpreted as `unsigned char` and then converted to `int`. This type is returned by functions, such as `fgetc`, `getc`, `getchar`, and `ungetc`, that read character data from a stream. As we've seen, this type is also accepted by the character-handling functions from `<ctype.h>`, because they might well be passed the result of `fgetc` or related functions.

wchar_t

The `wchar_t` type is an integer type added to C to process the characters of a large character set. It can be a signed or unsigned integer type, depending on the implementation, and has an implementation-defined inclusive range of `WCHAR_MIN` to `WCHAR_MAX`. Most implementations define `wchar_t` to be either a 16- or 32-bit unsigned integer type, but implementations that don't support localization may define `wchar_t` to have the same width as `char`. C does not permit a variable-length encoding for wide strings (despite UTF-16 being used in practice on Windows). Implementations can conditionally define the macro `__STDC_ISO_10646__` as an integer constant of the form *yyyymmL* (for example, `199712L`) to mean that the `wchar_t` type is used to represent Unicode characters corresponding to the specified version of the standard. Implementations that chose a 16-bit type for `wchar_t` cannot meet the requirements for defining `__STDC_ISO_10646__` for ISO/IEC 10646 editions

more recent than Unicode 3.1 (ISO/IEC 10646-1:2000 and ISO/IEC 10646-2:2001). Consequently, the requirement for defining `__STDC_ISO_10646__` is either a `wchar_t` type larger than 20 bits, or a 16-bit `wchar_t` and a value for `__STDC_ISO_10646__` earlier than `200103L`. The `wchar_t` type can be used for encodings other than Unicode, such as wide EBCDIC.

Writing portable code using `wchar_t` can be difficult because of the range of implementation-defined behavior. For example, Windows uses a 16-bit unsigned integer type, while Linux typically uses a 32-bit unsigned integer type. Code that calculates the lengths and sizes of wide-character strings is error-prone and must be performed with care.

char16_t and char32_t

Newer languages (including Ada95, Java, TCL, Perl, Python, and C#) have data types for Unicode characters. C11 introduced the 16- and 32-bit character data types `char16_t` and `char32_t`, declared in `<uchar.h>`, to provide data types for UTF-16 and UTF-32 encodings, respectively. C11 doesn't include library functions for the new data types, except for one set of character conversion functions, to allow library developers to implement them. Without library functions, these types have limited usefulness.

C defines two environment macros that indicate how characters represented in these types are encoded. If the environment macro `__STDC_UTF_16__` has the value `1`, values of type `char16_t` are UTF-16 encoded. If the environment macro `__STDC_UTF_32__` has the value `1`, values of type `char32_t` are UTF-32 encoded. If the macro isn't defined, another implementation-defined encoding is used. Visual C++ does not define these macros.

Character Constants

C allows you to specify *character constants*, also known as *character literals*, which are sequences of one or more characters enclosed in single quotes, such as `'ÿ'`. Character constants allow you to specify character values in the source code of your program. Table 7-1 shows the types of character constants that can be specified in C.

Table 7-1: Types of Character Constants

Prefix	Type
None	`int`
`L'a'`	The unsigned type corresponding to `wchar_t`
`u'a'`	`char16_t`
`U'a'`	`char32_t`

The oddest thing in Table 7-1 is that a nonprefixed character constant, such as `'a'`, has type `int` rather than type `char`. In C, for historical reasons, if a character constant contains only a single character or escape sequence, the value of the character constant is represented as an object with type

`char` converted to type `int`. This differs from C++, in which a character literal that contains only one character has type `char`.

The value of a character constant containing more than one character (for example, `'ab'`) is implementation defined. So is the value of a source character that cannot be represented as a single code unit in the execution character set. The earlier example of `'ÿ'` is one such case. If the execution character set is UTF-8, the value might be `0xC3BF` to reflect the UTF-8 encoding of the two code units needed to represent the `U+00FF` code-point value. C2x will add the `u8` prefix for character literals to represent a UTF-8 encoding. Until the release of C2x, there will not be a character-literal prefix for UTF-8 characters, unless implementations decide to implement it in advance.

Escape Sequences

The single quote (`'`) and backslash (`\`) have special meanings, so they cannot be directly represented as characters. Instead, to represent the single quote, we use the escape sequence `\'`, and to represent the backslash, we use `\\`. We can represent other characters, such as the question mark (`?`), and arbitrary integer values by using the escape sequences shown in Table 7-2.

Table 7-2: Escape Sequences

Character	Escape sequence
Single quote	`\'`
Double quote	`\"`
Question mark	`\?`
Backslash	`\\`
Alert	`\a`
Backspace	`\b`
Form feed	`\f`
Newline	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Octal character	`\`*`up to 3 octal digits`*
Hexadecimal character	`\x` *`hexadecimal digits`*

The following nongraphical characters are represented by escape sequences consisting of the backslash followed by a lowercase letter: `\a` (alert), `\b` (backspace), `\f` (form feed), `\n` (newline), `\r` (carriage return), `\t` (horizontal tab), and `\v` (vertical tab).

Octal digits can be incorporated into an octal escape sequence to construct a single character for a character constant, or a single wide character for a wide-character constant. The numerical value of the octal integer specifies the value of the desired character or wide character. *A backslash followed by numbers is always interpreted as an octal value.* For example, you can

represent the backspace character (8 decimal) as the octal value `\10` or, equivalently, `\010`.

We can also incorporate the hexadecimal digits that follow the `\x` to construct a single character or wide character for a character constant. The numerical value of the hexadecimal integer forms the value of the desired character or wide character. For example, you can represent the backspace character as the hexadecimal value `\x8` or, equivalently, `\x08`.

Linux

Character encodings have evolved differently on various operating systems. Before UTF-8 emerged, Linux typically relied on various language-specific extensions of ASCII. The most popular of these were ISO 8859-1 and ISO 8859-2 in Europe, ISO 8859-7 in Greece, KOI-8/ISO 8859-5/CP1251 in Russia, EUC and Shift-JIS in Japan, and BIG5 in Taiwan. Linux distributors and application developers are phasing out these older legacy encodings in favor of UTF-8 to represent localized text strings (Kuhn 1999).

GCC has several flags that allow you to configure character sets. Here are a couple of flags you may find useful:

```
-fexec-charset=charset
```

The `-fexec-charset` flag sets the execution character set that's used to interpret string and character constants. The default is UTF-8. The *charset* can be any encoding supported by the system's `iconv` library routine described later in this chapter. For example, setting `-fexec-charset=IBM1047` instructs GCC to interpret string constants hardcoded in source code, such as `printf` format strings, according to EBCDIC code page 1047.

To select the wide execution character set, used for wide string and character constants, use the `-fwide-exec-charset` flag:

```
-fwide-exec-charset=charset
```

The default is UTF-32 or UTF-16, corresponding to the width of `wchar_t`.

To set the input character set, used for translation from the character set of the input file to the source character set used by GCC, use the `-finput-charset` flag:

```
-finput-charset=charset
```

Clang has `-fexec-charset` and `-finput-charset`, but not `-fwide-exec-charset`. Clang allows you to set *charset* to UTF-8 only and rejects any attempt to set it to something else.

Windows

Support for character encodings in Windows has irregularly evolved. Programs developed for Windows can handle character encodings

using either Unicode interfaces or interfaces that implicitly rely on locale-dependent character encodings. For most modern applications, you should choose the Unicode interfaces by default to ensure that the application behaves as you expect when processing text. Generally, this code will have better performance as narrow strings passed to Windows library functions are frequently converted to Unicode strings.

The main and wmain Entry Points

Visual C++ supports two entry points to your program: `main`, which allows you to pass narrow-character arguments, and `wmain`, which allows you to pass wide-character arguments. You declare formal parameters to `wmain` by using a format similar to `main`, as shown in Table 7-3.

Table 7-3: Windows Program Entry Point Declarations

Narrow-character arguments	Wide-character arguments

`int main(void);`
`int main(int argc, char *argv[]);`
`int main(int argc, char *argv[],`
`  char *envp[]);`
	`int wmain(void);`
	`int wmain(int argc, wchar_t *argv[]);`
	`int wmain(int argc, wchar_t *argv[],`
	`  wchar_t *envp[]);`

For either entry point, the character encoding ultimately depends on the calling process. However, by convention, the `main` function generally receives its optional arguments and environment as pointers to text encoded with the current Windows (also called ANSI) code page, while the `wmain` function generally receives UTF-16 encoded text.

When you run a program from a shell such as the command prompt, the shell's command interpreter converts the arguments into the proper encoding for that entry point. A Windows process starts with a UTF-16 encoded command line. The startup code emitted by the compiler/linker calls the `CommandLineToArgvW` function to convert the command line to the `argv` form required to call `main`, or passes the command line arguments directly to the `argv` form required to call `wmain`. In a call to `main`, the results are then transcoded to the current Windows code page, which can vary from system to system. The ASCII character `?` is substituted for characters that lack representation in the current Windows code page.

The Windows console uses an original equipment manufacturer (OEM) code page when writing data to the console. The actual encoding used varies from system to system, but is often different from the Windows code page. For example, on a US English version of Windows, the Windows code page may be Windows Latin 1, while the OEM code page may be DOS Latin US. In general, writing textual data to `stdout` or `stderr` requires the text to be converted to the OEM code page first, or requires setting the console's output code page to match the encoding of the text being written out. Failure to do so may cause unexpected output to be printed to the console. However, even if you carefully match the character encodings between your program and the console, the console might still fail to display the characters as expected because of other factors, such as the current font selected

for the console not having the appropriate glyphs required to represent the characters. Additionally, the Windows console has historically been unable to display characters outside of the Unicode BMP because it stores only a 16-bit value for character data for each cell.

Narrow versus Wide Characters

There are two versions of all system APIs in the Win32 SDK: a narrow Windows (ANSI) version with an `A` suffix, and a wide character version with a `W` suffix:

```
int SomeFuncA(LPSTR SomeString);
int SomeFuncW(LPWSTR SomeString);
```

You should determine whether your app is going to use wide (UTF-16) or narrow characters and then code accordingly. The best practice is to explicitly call the narrow or wide string version of each function and pass a string of the appropriate type:

```
SomeFuncW(L"String");
SomeFuncA("String");
```

Examples of actual functions from the Win32 SDK include the `MessageBoxA`/`MessageBoxW` and `CreateWindowExA`/`CreateWindowExW` functions.

Character Conversion

Although international text is increasingly encoded in Unicode, it is still encoded in language- or country-dependent character encodings, making it necessary to convert between these encodings. In particular, Windows still operates in locales with traditional, limited character encodings, such as IBM EBCDIC and ISO 8859-1. Programs frequently need to convert between the Unicode and traditional encoding schemes when performing I/O.

It is not possible to convert all strings to each language- or country-dependent character encoding. This is obvious when the encoding is US-ASCII, which can't represent any character requiring more than seven bits of storage. Latin-1 will never encode the character 晥 properly, and many kinds of non-Japanese letters and words cannot be converted to Shift-JIS without losing information.

The following sections describe various approaches to converting between character encodings.

C Standard Library

The C Standard Library provides a handful of functions to convert between narrow code units (`char`) and wide code units (`wchar_t`). The `mbtowc` (multibyte to wide character), `wctomb` (wide character to multibyte), `mbrtowc` (multibyte restartable to wide character), and `wcrtomb` (wide character restartable to multibyte) functions convert one code unit at a time, writing the result to an output object or buffer. The `mbstowcs` (multibyte

string to wide character string), `wcstombs` (wide character string to multibyte string), `mbsrtowcs` (multibyte string restartable to multibyte string), and `wcsrtombs` (wide character string restartable to multibyte string) functions convert strings of code units at a time, writing the result to an output buffer.

Conversion functions need to store data to properly process a sequence of conversions between function calls. The *nonrestartable* forms store the state internally. The *restartable* versions have an additional parameter that is a pointer to an object of type `mbstate_t` that describes the current conversion state of the associated multibyte character sequence. This object holds the state data that makes it possible to restart the conversion where it left off after another call to the function to perform an unrelated conversion. The *string* versions are for performing bulk conversions of multiple code units at once.

These functions have a few limitations. As discussed earlier, Windows uses 16-bit code units for `wchar_t`. This can be a problem, because the C Standard requires an object of type `wchar_t` to be capable of representing any character in the current locale, and a 16-bit code unit can be too small to do so. C technically doesn't allow you to use multiple objects of type `wchar_t` to represent a single character. Consequently, the standard conversion functions may result in a loss of data. On the other hand, most POSIX implementations use 32-bit code units for `wchar_t`, allowing the use of UTF-32. Because a single UTF-32 code unit can represent a whole code point, conversions using standard functions cannot lose or truncate data.

The C Standards committee added the following functions to C11 to address the potential loss of data using standard conversion functions:

`mbrtoc16`, `c16rtomb` Converts between a sequence of narrow code units and one or more `char16_t` code units

`mbrtoc32`, `c32rtomb` Converts a sequence of narrow code units to one or more **`char32_t`** code units

The first two functions convert between locale-dependent character encodings, represented as an array of `char`, and UTF-16 data stored in an array of `char16_t` (assuming `__STDC_UTF_16__` has the value `1`). The second two functions convert between the locale-dependent encodings and UTF-32 data stored in array of `char32_t` encoded data (assuming `__STDC_UTF_32__` has the value `1`). The program shown in Listing 7-1 uses the `mbrtoc16` function to convert a UTF-8 input string to a UTF-16 encoded string.

```
#include <locale.h>
#include <uchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

❶ #if __STDC_UTF_16__ != 1
  #error "__STDC_UTF_16__ not defined"
```

```
#endif

int main(void) {
❷ setlocale(LC_ALL, "en_US.utf8");
  char input[] = u8"I ♥ 🦔s!";
  const size_t input_size = sizeof(input);
  char16_t output[input_size]; // UTF-16 requires less code units than UTF-8
  char *p_input = input;
  char *p_end = input + input_size;
  char16_t *p_output = output;
  size_t code;
  mbstate_t state = {0};
  puts(input);
❸ while ((code = mbrtoc16(p_output, p_input, p_end-p_input, &state))){
    if (code == (size_t)-1)
      break; // -1 - invalid code unit sequence detected
    else if (code == (size_t)-2)
      break; // -2 - code unit sequence missing elements
    else if (code == (size_t)-3)
      p_output++; // -3 - high surrogate from a surrogate pair
    else {
      p_output++;  // one value was written out
      p_input += code;  // code is the # of code units read by function
    }
  }
  size_t output_size = p_output - output + 1;
  printf("Converted to %zu UTF-16 code units: [ ", output_size);
  for(size_t x = 0; x < output_size; ++x) printf("%#x ", output[x]);
  puts("]");
}
```

Listing 7-1: Using the `mbrtoc16` function to convert a UTF-8 string to a `char16_t` string

We call the `setlocale` function ❷ to set the multibyte character encoding to UTF-8 by passing an implementation-defined string. The preprocessing directives ❶ ensure that the macro `__STDC_UTF_16__` has the value `1`. (Refer to Chapter 9 for more information on preprocessing directives.) As a result, each call to the `mbrtoc16` function converts a single code point from a UTF-8 representation to a UTF-16 representation. If the resulting UTF-16 code unit is a high surrogate (from a surrogate pair), the `state` object is updated to indicate that the next call to `mbrtoc16` will write out the low surrogate without considering the input string.

There is no string version of the `mbrtoc16` function, so we loop through a UTF-8 input string iteratively, calling the `mbrtoc16` function ❸, to convert it to a UTF-16 string. In the case of an encoding error, the `mbrtoc16` function returns `(size_t)-1` and, if the code unit sequence is missing elements, it returns `(size_t)-2`. If either situation occurs, the loop terminates and the conversion ends.

A return value of `(size_t)-3` means that the function output the high surrogate from a surrogate pair, and then stored an indicator in the state parameter. The indicator is used the next time the `mbrtoc16` function is called so it can output the low surrogate from a surrogate pair to form a

complete `char16_t` sequence that represents a single code point. All restartable encoding conversion functions in the C Standard behave similarly with the state parameter.

If the function returns anything other than `(size_t)-1`, `(size_t)-2`, or `(size_t)-3`, the `p_output` pointer is incremented and the `p_input` pointer is increased by the number of code units read by the function, and the conversion of the string continues.

libiconv

GNU `libiconv` is a commonly used cross-platform, open source library for performing string encoding conversions. It includes the `iconv_open` function that allocates a conversion descriptor you can use to convert byte sequences from one character encoding to another. The documentation for this function, found at *https://www.gnu.org/*, defines strings you can use to identify a particular *`charset`* such as `ASCII`, `ISO-8859-1`, `SHIFT_JIS`, or `UTF-8` to denote the locale-dependent character encoding.

Win32 Conversion APIs

The Win32 SDK provides two functions for converting between wide and narrow character strings:

`MultiByteToWideChar` Maps a character string to a new UTF-16 (wide character) string

`WideCharToMultiByte` Maps a UTF-16 (wide character) string to a new character string

The `MultiByteToWideChar` function maps string data that's encoded in an arbitrary character code page to a UTF-16 string. Similarly, the function `WideCharToMultiByte` maps string data encoded in UTF-16 to an arbitrary character code page. Because UTF-16 data cannot be represented by all code pages, this function can specify a default character to use in place of any UTF-16 character that cannot be converted.

Strings

C doesn't support a primitive string type and probably never will. Instead, it implements strings as arrays of characters. C has two types of strings: narrow and wide.

A *narrow string* has the type array of `char`. It consists of a contiguous sequence of characters that include a terminating null character. A pointer to a string points to its initial character. The size of a string is the number of bytes allocated to the backing array storage. The length of a string is the number of code units (bytes) preceding the first null character. In Figure 7-1, the size of the string is 7, and the length of the string is 5. Elements of the backing array beyond the last element must not be accessed. Elements of the array that haven't been initialized must not be read.

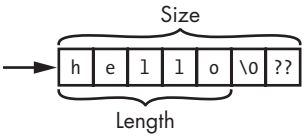

Figure 7-1: Sample narrow string

A *wide string* has the type array of `wchar_t`. It's a contiguous sequence of wide characters that include a terminating null wide character. A *pointer* to a wide string points to its initial wide character. The *length* of a wide string is the number of code units preceding the first null wide character. Figure 7-2 illustrates both the UTF-16BE (big endian) and UTF-16LE (little endian) representation of *hello*. The size of the array is implementation-defined. This array is 14 bytes and assumes an implementation that has an 8-bit byte and 16-bit `wchar_t` type. The length of this string is 5, as the number of characters has not changed. Elements of the backing array beyond the last element must not be accessed. Elements of the array that haven't been initialized must not be read.

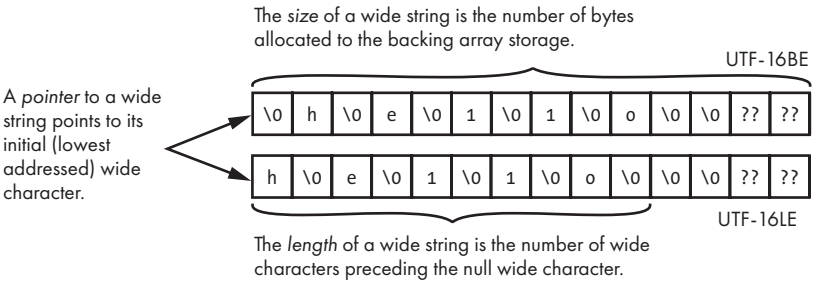

Figure 7-2: Sample UTF-16LE wide string

String Literals

A *character string literal* is a string constant represented by a sequence of zero or more multibyte characters enclosed in double quotes; for example, `"ABC"`. You can use various prefixes to declare string literals of different character types:

- `char` string literal type, such as `"ABC"`
- `wchar_t` string literal type with `L` prefix, such as `L"ABC"`
- UTF-8 string literal type with `u8` prefix, such as `u8"ABC"`
- `char16_t` string literal type with `u` prefix, such as `u"ABC"`
- `char32_t` string literal type with `U` prefix, such as `U"ABC"`

The C Standard doesn't mandate that an implementation use ASCII for string literals. However, you can use the `u8` prefix to force a string literal to be UTF-8 encoded, and if all the characters in the literal are ASCII

characters, the compiler will produce an ASCII string literal, even if the implementation would normally encode string literals in another encoding (for example, EBCDIC).

A string literal has a non-`const` array type. Modifying a string literal is undefined behavior and prohibited by the CERT C rule STR30-C (Do not attempt to modify string literals). This is because these string literals may be stored in read-only memory, or multiple string literals may share the same memory, resulting in multiple strings being altered if one string is modified.

String literals often initialize array variables, which you can declare with an explicit bound that matches the number of characters in the string literal. Consider the following declaration:

```
#define S_INIT "abc"
// ---snip---
const char s[4] = S_INIT;
```

The size of the array `s` is four, the exact size required to initialize the array to the string literal, including the space for a trailing null byte.

If you add another character to the string literal used to initialize the array, however, the meaning of the code changes substantially:

```
#define S_INIT "abcd"
// ---snip---
const char s[4] = S_INIT;
```

The size of the array `s` remains four, although the size of the string literal is now five. As a result, the array `s` is initialized to the character array `"abcd"` with the trailing null byte omitted. This syntax is by design, to allow you to initialize a character array and not a string. Therefore, it is unlikely that your compiler will diagnose this declaration as an error. GCC issues a warning when such an array is passed as an argument to a string function.

There is some risk that, if the string literal changes during maintenance, a string could unintentionally be changed to a character array with no terminating null character, particularly when the string literal is defined separately from the declaration, as in this example. If your intent is to always initialize `s` to a string, you should omit the array bound. If you don't specify the bound of the array, the compiler will allocate sufficient space for the entire string literal, including the terminating null character:

```
const char s[] = S_INIT;
```

This approach simplifies maintenance because the size of the array can always be determined even if the size of the string literal changes.

The size of arrays declared using this syntax can be determined at compile time by using the `sizeof` operator:

```
size_t size = sizeof s;
```

If, instead, we declared this string as follows

```
const char *foo = S_INIT;
```

we would need to invoke the `strlen` function to get the following length

```
size_t size = strlen(foo) + 1U;
```

which may incur a runtime cost.

String-Handling Functions

Several approaches can be used to manage strings in C, the first of which are the C Standard Library functions. Narrow string-handling functions are defined in the `<string.h>` header file, and wide string-handling functions in `<wchar.h>`. These legacy string-handling functions have been associated in recent years with various security vulnerabilities. This is because they don't check the size of the array (frequently lacking the information needed to perform such checks) and trust you to provide adequately sized character arrays to hold the output. While it is possible to write safe, robust, and error-free code using these functions, the library promotes programming styles that can result in buffer overflows if a result is too large for the provided array. These functions aren't inherently insecure but are prone to misuse and need to be used carefully (or not at all).

As a result, C11 introduced the normative (but optional) Annex K bounds-checking interfaces. This annex provides alternative library functions intended to promote safer, more secure programming by requiring you to provide the length of output buffers, for example, and validating that these buffers are adequately sized to contain the output from these functions. For instance, Annex K defines the `strcpy_s`, `strcat_s`, `strncpy_s`, and `strncat_s` functions as close replacements for the C Standard Library's `strcpy`, `strcat`, `strncpy`, and `strncat` functions.

POSIX also defines several string-handling functions, such as `strdup` and `strndup`, that provide another set of string APIs you can use on POSIX-compliant platforms such as Linux and Unix (IEEE Std 1003.1:2018).

Visual C++ provides all the string-handling functions defined by the C Standard Library up to C99 but doesn't implement the full POSIX specification.[10] Visual C++ names many of its runtime library functions with a leading underscore; for example, `_strdup` instead of `strdup`. Visual C++ also supports many of the safe string-handling functions from Annex K and will diagnose use of an unsafe variant, unless you define `_CRT_SECURE_NO_WARNINGS` prior to including the header file that declares the function.

We'll discuss each of these string-handling libraries in the subsequent sections.

10. For more information on Microsoft's POSIX support, see Compatibility at *https://docs.microsoft.com/en-us/cpp/c-runtime-library/compatibility?view=vs-2019/*.

<string.h> and <wchar.h>

The C Standard Library includes well-known functions such as `strcpy`, `strncpy`, `strcat`, `strncat`, `strlen`, and so forth, as well as the `memcpy` and `memmove` functions you can use to copy and move strings, respectively. The C Standard also provides a wide character interface that operates on objects of type `wchar_t` instead of `char`. (These function names are similar to the narrow string function names, except that `str` is replaced with `wc`, and a `w` is added in front of the memory function names.) Table 7-4 gives some examples of narrow and wide character string functions. It's important not to confuse them.

Table 7-4: Narrow and Wide String Functions

Narrow (`char`)	Wide (`wchar_t`)	Description
`strcpy`	`wcscpy`	String copy
`strncpy`	`wcsncpy`	Truncated, zero-filled copy
`memcpy`	`wmemcpy`	Copies a specified number of code units
`memmove`	`wmemmove`	Copies a specified number of (possibly overlapping) code units
`strcat`	`wcscat`	Concatenates strings
`strncat`	`wcsncat`	Concatenates strings with truncation
`strcmp`	`wcscmp`	Compares strings
`strncmp`	`wcsncmp`	Compares strings to a point
`strchr`	`wcschr`	Locates a character in a string
`strcspn`	`wcscspn`	Computes the length of a complementary string segment
`strpbrk`	`wcspbrk`	Finds the first occurrence of a set of characters in a string
`strrchr`	`wcsrchr`	Finds the first occurrence of a character in a string
`strspn`	`wcsspn`	Computes the length of a string segment
`strstr`	`wcsstr`	Finds a substring
`strtok`	`wcstok`	String tokenizer (modifies the string being tokenized)
`memchr`	`wmemchr`	Finds a code unit in memory
`strlen`	`wcslen`	Computes string length
`memset`	`wmemset`	Fills memory with a specified code unit

These string-handling functions are considered efficient because they leave memory management to the caller and can be used with both statically and dynamically allocated storage. In the next couple of sections, I'll go into more detail on some of the most commonly used of these functions.

Size and Length

As mentioned earlier in this chapter, strings have both a *size* (which is the number of bytes allocated to the backing array storage) and a *length*. You can determine the size of a statically allocated backing array at compile time by using the `sizeof` operator:

```
char str[100] = "Here comes the sun";
size_t str_size = sizeof(str); // is 100
```

You can compute the length of a string by using the `strlen` function:

```
char str[100] = "Here comes the sun";
size_t str_len = strlen(str); // str_len is 18
```

The `wcslen` function computes the length of a wide string measured by the number of code units preceding the terminating null wide character:

```
wchar_t str[100] = L"Here comes the sun";
size_t str_len = wcslen(str); // str_len is 18
```

The length is a count of something, but what exactly is being counted can be unclear. Here are some of the things that *could* be counted when taking the length of a string:

Bytes Useful when allocating storage.

Code units Number of individual code units used to represent the string. This length depends on encoding and can also be used to allocate memory.

Code points Code points (characters) can take up multiple code units. This value is not useful when allocating storage.

Extended grapheme cluster A group of one or more Unicode scalar values that approximates a single user-perceived character. Many individual characters, such as "é", "김", and "😎" may be constructed from multiple Unicode scalar values. Unicode's boundary algorithms combine these code points into extended grapheme clusters.

The `strlen` and `wcslen` functions count code units. For the `strlen` function, this corresponds to the number of bytes. Determining the amount of storage required by using the `wcslen` function is more complicated because the size of the `wchar_t` type is implementation defined. Listing 7-2 contains examples of dynamically allocating storage for both narrow and wide strings.

```
// narrow strings
char str1[] = "Here comes the sun";
char *str2 = malloc(strlen(str1) + 1));
```

```
// wide strings
wchar_t wstr1[] = L"Here comes the sun";
wchar_t *wstr2 = malloc((wcslen(wstr1) + 1) * sizeof(wchar_t));
```

Listing 7-2: Dynamically allocating storage for narrow and wide string functions

For narrow strings, we can determine the size of the string by using the `strlen` function and then adding `1` for the terminating null character before allocating. For wide strings, we can determine the size of the string by using the `wcslen` function, add `1` to accommodate the terminating null wide character, and then multiply the result of this operation by the size of the `wchar_t` type.

Code point or extended grapheme clusters counts cannot be used for storage allocation, because they consist of an unpredictable number of code units.[11] Extended grapheme clusters are used to determine where to truncate a string, for example, because of a lack of storage. Truncation at extended grapheme cluster boundaries avoids slicing user-perceived characters.

Calling the `strlen` function can be an expensive operation because it needs to traverse the length of the array looking for a null character. The following is a straightforward implementation of the `strlen` function:

```
size_t strlen(const char * str) {
  const char *s;
  for (s = str; *s; ++s) {}
  return s - str;
}
```

The `strlen` function has no way of knowing the size of the object referenced by `str`. If you call `strlen` with an invalid string that lacks a null character before the bound, the function will access the array beyond its end, resulting in undefined behavior. Passing a null pointer to `strlen` will also result in undefined behavior (a null-pointer dereference). This implementation of the `strlen` function also has undefined behavior for strings larger than `PTRDIFF_MAX`. You should refrain from creating such objects (in which case this implementation is fine).

The strcpy Function

Calculating the size of dynamically allocated memory is not as easy. One approach is to store the size when allocating and reuse this value later. The code snippet in Listing 7-3 uses the `strcpy` function to make a copy of `str` by determining the length and then adding `1` to accommodate the terminating null character:

```
  char str[100] = "Here comes the sun";
  size_t str_size =  strlen(str) + 1;
```

11. For an interesting exposition on string length, see "It's Not Wrong that "🤦".length == 7" at *https://hsivonen.fi/string-length/*.

```
  char *dest = (char *)malloc(str_size);
  if (dest) {
    strcpy(dest, str);
  }
  else {
    /* handle error */
  }
```

Listing 7-3: Copying a string

We can then use `str_size` to dynamically allocate the storage for the copy. The `strcpy` function copies the string from the source string (`str`) to the destination string (`dest`), including the terminating null character. The `strcpy` function returns the address of the beginning of the destination string, which is ignored in this example.

A common implementation of the `strcpy` function is as follows:

```
char *strcpy(char *dest, const char *src) {
  char *save = dest;
  while (*dest++ = *src++);
  return save;
}
```

This code saves a pointer to the destination string in `save` (to use as the return value) before copying all the bytes from the source to the destination array. The `while` loop terminates when the first null byte is copied. Because `strcpy` doesn't know the length of the source string or the size of the destination array, it assumes that all the function's arguments have been validated by the caller, allowing the implementation to simply copy each byte from the source string to the destination array without checks.

Argument Checking

Argument checking can be performed by either the calling function or the called function. Redundant argument testing by both the caller and the callee is a largely discredited style of defensive programming. The usual discipline is to require validation on only one side of each interface.

The most time-efficient approach is for the caller to perform the check, because the caller should have a better understanding of the program state. In Listing 7-3, we can see that the arguments to `strcpy` are valid without introducing further redundant tests: the variable `str` references a statically allocated array that was properly initialized in the declaration, and the `dest` parameter is a non-null pointer referencing dynamically allocated storage of sufficient size to hold a copy of `str`, including the null character. Therefore, the call to `strcpy` is safe, and the copy can be performed in a time-efficient manner. This approach to argument checking is commonly used by C Standard Library functions because it adheres to the "spirit of C," in that it's optimally efficient and trusts the programmer to pass valid arguments.

The safer, more secure, space-efficient approach is for the callee to check the arguments. This approach is less error-prone because the library function implementer validates the arguments, so we no longer need to trust the programmer to pass valid ones. The function implementer is usually in a better position to understand which arguments need to be validated. If the input validation code is defective, the repair needs to be made in only one place. All the code to validate the arguments is in one place, so this approach is typically more space-efficient. However, because these tests run even when unnecessary, they can also be less time-efficient. Frequently, the caller of these functions will place checks before suspect system calls that may or may not already perform similar checks. This approach would also impose additional error handling on callees that don't currently return error indications but would presumably need to if they validated arguments. For strings, the called function can't always determine whether the argument is a valid null-terminated string, or points to sufficient space to make a copy.

The lesson here is don't assume that the C Standard Library functions validate arguments unless the standard explicitly requires them to.

The memcpy Function

The `memcpy` function copies a specified number of characters, `size`, from the object referenced by `src` into the object referenced by `dest`:

```
void *memcpy(void * restrict dest, const void * restrict src, size_t size);
```

You can use the `memcpy` function instead of `strcpy` to copy strings when the size of the destination array is larger than or equal to the `size` argument to `memcpy`, the source array contains a null character before the bound, and the string length is less than `size - 1` (so that the resulting string will be properly null-terminated). The best advice is to use `strcpy` when copying a string, and `memcpy` when copying only raw, untyped memory. Also remember that the assignment (`=`) operator can efficiently copy objects in many cases.

Most of the C Standard Library functions return a pointer to the beginning of the string passed as an argument so that you can nest calls to string functions. For example, the following sequence of nested function calls constructs a person's full legal name by copying, then concatenating, the constituent parts:

```
strcat(strcat(strcat(strcat(strcpy(full, first), " "), middle), " "), last);
```

However, piecing together the array `full` from its substrings requires this string to be scanned many more times than necessary; it would have been more useful for the functions to return pointers to the *end* of the modified string, to eliminate this need for rescanning. C2x will introduce a string-copy function with a better interface design, `memccpy`. POSIX environments should already provide this, but you may need to enable its declaration as follows:

```
#define _XOPEN_SOURCE 700
#include <string.h>
```

The gets Function

The gets function is a flawed input function that accepts input without providing any way to specify the size of the destination array. For that reason, it cannot prevent buffer overflows. As a result, the gets function was deprecated in C99 and eliminated from C11. However, it has been around for many years, and most libraries still provide an implementation for backward compatibility, so you may see it in the wild. You should *never* use this function, and you should replace any use of the gets function you find in any code you are maintaining.

Because the gets function is so bad, we'll spend some time examining why it's so awful. The function shown in Listing 7-4 prompts the user to enter either y or n to indicate whether they'd like to continue. This function has undefined behavior if more than eight characters are entered at the prompt. However, the gets function has no way of knowing how large the destination array is and will simply write beyond the end of the array object.

```
#include <stdio.h>
#include <stdlib.h>
void get_y_or_n(void) {
  char response[8];
  puts("Continue? [y] n: ");
  gets(response);
  if (response[0] == 'n')
    exit(0);
  return;
}
```

Listing 7-4: Misuse of the obsolete gets function

Listing 7-5 shows a simplified implementation of the gets function. As you can see, the caller of this function has no way to limit the number of characters read.

```
char *gets(char *dest) {
  int c;
  char *p = dest;
  while ((c = getchar()) != EOF && c != '\n') {
    *p++ = c;
  }
  *p = '\0';
  return dest;
}
```

Listing 7-5: gets function implementation

The gets function iterates reading a character at a time. The loop terminates if either an EOF or newline '\n' character is read. Otherwise, the function will continue to write to the dest array without concern for the boundaries of the object.

Listing 7-6 shows the `get_y_or_n` function from Listing 7-4 with the `gets` function inlined.

```
#include <stdio.h>
#include <stdlib.h>
void get_y_or_n(void) {
  char response[8];
  puts("Continue? [y] n: ");
  int c;
  char *p = response;
❶ while ((c = getchar()) != EOF && c != '\n') {
    *p++ = c;
  }
  *p = '\0';
  if (response[0] == 'n')
    exit(0);
}
```

Listing 7-6: Poorly written `while` loop

The size of the destination array is now available, but the `while` loop ❶ doesn't use this information. You should ensure that reaching the size of the destination array is a loop termination condition when reading or writing to an array in a loop such as this one.

Annex K Bounds-Checking Interfaces

C11 introduced the Annex K bounds-checking interfaces with alternative functions that verify that output buffers are large enough for the intended result and return a failure indicator if they aren't. These functions are designed to prevent writing data past the end of an array, and to null-terminate all string results. These string-handling functions leave memory management to the caller, and memory can be statically or dynamically allocated before the functions are invoked.

Microsoft created the C11 Annex K functions to help retrofit its legacy code base in response to numerous, well-publicized security incidents in the 1990s. These functions were then proposed to the C Standards committee for standardization, published as ISO/IEC TR 24731-1 (ISO/IEC TR 24731-1:2007), and then later incorporated into C11 as a set of optional extensions. Despite the improved usability and security provided by these functions, they aren't yet widely implemented at the time of writing.

The gets_s Function

The Annex K bounds-checking interface has a `gets_s` function we can use to eliminate the undefined behavior caused by the `gets` function in Listing 7-4, as shown in Listing 7-7. The two functions are similar, except that the `gets_s` function checks the array bounds. The default behavior that occurs when the maximum number of characters input is exceeded is implementation

defined, but typically the abort function is called. You can change this behavior via the set_constraint_handler_s function, which I'll explain further in "Runtime Constraints" on page 143.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <stdlib.h>

void get_y_or_n(void) {
  char response[8];
  size_t len = sizeof(response);
  puts("Continue? [y] n: ");
  gets_s(response, len);
  if (response[0] == 'n') exit(0);
}
```

Listing 7-7: Use of the gets_s function.

The first line in Listing 7-7 defines the __STDC_WANT_LIB_EXT1__ macro to expand to the value 1. We then include the header files that define the bounds-checking interfaces, allowing them to be used in your program. Unlike the gets function, the gets_s function takes a size argument. Consequently, the revised function calculates the size of the destination array by using the sizeof operator and passes this value as an argument to the gets_s function. The implementation-defined behavior is the result of the runtime-constraint violation.

The strcpy_s Function

The strcpy_s function is a close replacement for the strcpy function defined in <string.h>. The strcpy_s function copies characters from a source string to a destination character array up to and including the terminating null character. Here's the strcpy_s signature:

```
errno_t strcpy_s(
  char * restrict s1, rsize_t s1max, const char * restrict s2
);
```

The strcpy_s function has an extra argument of type rsize_t that specifies the maximum length of the destination buffer. The strcpy_s function succeeds only when it can fully copy the source string to the destination without overflowing the destination buffer. The strcpy_s function verifies that the following runtime constraints aren't violated:

- Neither s1 nor s2 are null pointers.
- s1max is not > RSIZE_MAX.
- s1max does not equal zero.
- s1max is > strnlen_s(s2, s1max).
- Copying does not take place between objects that overlap.

To perform the string copy in a single pass, a typical `strcpy_s` function implementation retrieves a character from the source string and copies it to the destination array until it has copied the entire string or the destination array is full. If it can't copy the entire string and `s1max` is positive, the `strcpy_s` function sets the first byte of the destination array to the null character, creating an empty string.

Runtime Constraints

Runtime constraints are violations of a function's runtime requirements that the function will detect and diagnose by a call to a handler. If this handler returns, the functions will return a failure indicator to the caller.

The bounds-checking interfaces enforce runtime constraints by invoking a runtime-constraint handler, which may simply return. Alternatively, the runtime-constraint handler might print a message to `stderr` and/or abort the program. You can control which handler function is called via the `set_constraint_handler_s` function, and can make the handler simply return as follows:

```
int main(void) {
  constraint_handler_t oconstraint =
    set_constraint_handler_s(ignore_handler_s);
  get_y_or_n();
}
```

If the handler returns, the function that identified the runtime-constraint violation and invoked the handler indicates a failure to its caller by using its return value.

The bounds-checking interface functions typically check the conditions immediately upon entry, or as they perform their tasks and gather sufficient information to determine whether a runtime constraint has been violated. The runtime constraints of the bounds-checking interfaces are conditions that would otherwise be undefined behavior for C Standard Library functions.

Implementations have a default constraint handler that they invoke if no calls to the `set_constraint_handler_s` function have been made. The default handler's behavior may cause the program to exit or abort, but implementations are encouraged to provide reasonable behavior by default. This allows, for example, compilers customarily used to implement safety-critical systems to not abort by default. You must check the return value of calls to functions that can return and not simply assume their results are valid. Implementation-defined behavior can be eliminated by invoking the `set_constraint_handler_s` function before invoking any bounds-checking interfaces or using any mechanism that invokes a runtime-constraint handler.

Annex K provides the `abort_handler_s` and `ignore_handler_s` functions, which represent two common strategies for handling errors. The C implementation's default handler need not be either of these handlers.

POSIX

POSIX also defines several string-handling functions, such as `strdup` and `strndup` (IEEE Std 1003.1:2018), that provide another set of string-related APIs for POSIX-compliant platforms. The C Standards committee published these interfaces as Technical Report 24731-2 (ISO/IEC TR 24731-2:2010), although these interfaces have not yet been included in the C Standard.

These replacement functions use dynamically allocated memory to ensure that buffer overflows don't occur, and they implement a *callee allocates, caller frees* model. Each function ensures that enough memory is available (except when a call to `malloc` fails). The `strdup` function, for example, returns a pointer to a new string that contains a duplicate of the argument. The returned pointer should be passed to the C Standard `free` function to reclaim the storage when it's no longer needed.

Listing 7-8 contains a code snippet that uses the `strdup` function to make a copy of the string returned by the `getenv` function.

```
const char *temp = getenv("TMP");
if (temp != NULL) {
  char *tmpvar = strdup(temp);
  if (tmpvar != NULL) {
    printf("TMP = %s.\n", tmpvar);
    free(tmpvar);
  }
}
```

Listing 7-8: Copying a string by using the `strdup` function

The C Standard Library `getenv` function searches an environment list, provided by the host environment, for a string that matches the string referenced by a specified name (`"TMP"` in this example). Strings in this environment list are referred to as *environment variables* and provide an additional mechanism for communicating strings to a process. These strings don't have a well-defined encoding but typically match the system encoding used for command line arguments, `stdin`, and `stdout`.

The returned string (the value of the variable) may be overwritten by a subsequent call to the `getenv` function, so it's a good idea to retrieve any environmental variable you need before creating any threads to eliminate the possibility of a race condition. If later use is anticipated, you should copy the string so the copy can be safely referenced as needed, as illustrated by the idiomatic example shown in Listing 7-8.

The `strndup` function is equivalent to `strdup`, except that `strndup` copies at most `n` plus 1 bytes into the newly allocated memory (while `strdup` copies the entire string) and ensures that the newly created string is always properly terminated.

These POSIX functions can help prevent buffer overflows by automatically allocating storage for the resulting strings, but this requires introducing additional calls to `free` when this storage is no longer needed. This

means matching a call to `free` to each call to `strdup` or `strndup`, for example, which can be confusing to programmers who are more familiar with the behavior of the string functions defined by `<string.h>`.

Microsoft

Microsoft implements most of the C Standard Library functions, as well as parts of the POSIX standard. However, sometimes the Microsoft implementation of these APIs differs from the requirements of a given standard or has a function name that conflicts with an identifier reservation in another standard. In these circumstances, Microsoft will often prefix the function name with an underscore. For instance, the POSIX function `strdup` isn't available on Windows, but the function `_strdup` is available and behaves the same way.

The Visual C++ library includes the prototype implementation of the bounds-checking interfaces. Unfortunately, Visual C++ does not conform to C11 or TR 24731-1, because Microsoft chose not to update its implementation based on changes to the APIs that occurred during the standardization process. For example, Visual C++ doesn't provide the `set_constraint_handler_s` function but instead retains an older function with similar behavior but an incompatible signature:

```
_invalid_parameter_handler _set_invalid_parameter_handler(_invalid_parameter_handler)
```

Microsoft also doesn't define the `abort_handler_s` and `ignore_handler_s` functions, the `memset_s` function (which was not defined by TR 24731-1), or the `RSIZE_MAX` macro. Visual C++ also doesn't treat overlapping source and destination sequences as runtime-constraint violations and instead has undefined behavior in such cases. My NCC Group whitepaper "Bounds-Checking Interfaces: Field Experience and Future Directions" provides additional information on all aspects of the bounds-checked interfaces, including Microsoft's implementation (Seacord 2019).

Summary

In this chapter, you learned about character encodings, such as ASCII and Unicode. You also learned about the various data types used to represent characters in C language programs, such as `char`, `int`, `wchar_t`, and so forth. We then covered character conversion libraries, including C Standard Library functions, `libiconv`, and Windows APIs.

In addition to characters, you also learned about strings and the legacy functions and bounds-checked interfaces defined in the C Standard Library for handling strings, as well as some POSIX- and Microsoft-specific functions.

In the next chapter, you'll learn about input/output.

8

INPUT/OUTPUT

This chapter will teach you how to perform input/output (I/O) operations to read data from, or write data to, terminals and filesystems. I/O involves all the ways information enters or exits a program, without which your programs would be useless. We'll cover techniques that use C Standard streams and POSIX file descriptors. We'll start by discussing C Standard text and binary streams. We'll then cover different ways of opening and closing files using C Standard Library and POSIX functions.

Next we'll discuss reading and writing characters and lines, reading and writing formatted text, and reading to and from binary streams. We'll also cover stream buffering, stream orientation, and file positioning.

Many other devices and I/O interfaces (such as `ioctl`) are available but are not discussed in this chapter.

Standard I/O Streams

The C Standard specifies the use of streams to communicate with terminals and files stored on supported, structured storage devices. A *stream* is a uniform abstraction for communicating with files and devices that consume or produce sequential data such as sockets, keyboards, USB ports, and printers.

C uses the opaque `FILE` data type to represent streams. A `FILE` object holds the internal state information for the connection to the associated file, including the file position indicator, buffering information, an error indicator, and an end-of-file indicator. You should never allocate a `FILE` object yourself. C Standard Library functions operate on objects of type `FILE *` (that is, a pointer to the `FILE` type). As a result, streams are frequently referred to as *file pointers*.

The C Standard provides an extensive API, found in `<stdio.h>`, for operating on streams; we'll explore this API later in this chapter. However, because these I/O functions need to work with a wide variety of devices and filesystems across many platforms, they're highly abstracted, which makes them unsuitable for anything beyond the simplest applications.

For example, the C Standard has no concept of directories, because it must be able to work with nonhierarchical filesystems. The C Standard makes few references to filesystem-specific details, like file permissions or locking. However, function specifications frequently state that certain behaviors happen "to the extent that the underlying system supports it," meaning that they will occur only if they're supported by your implementation.

As a result, you'll generally need to use the less portable APIs provided by POSIX, Windows, and other platforms to perform I/O in real-world applications. Frequently, applications will define their own APIs that, in turn, rely on platform-specific APIs to provide safe and secure I/O operations across platforms.

Stream Buffering

Buffering is the process of temporarily storing data in main memory that's passing between a process and a device or file. Buffering improves the throughput of I/O operations, which often have high latencies. Similarly, when a program requests to write to block-oriented devices like disks, the driver can cache the data in memory until it has accumulated enough data for one or more device blocks, at which point it writes the data all at once to the disk, improving throughput. This strategy is called *flushing* the output buffer.

Like device drivers, streams often maintain their own I/O buffers. Typically, a stream uses one input buffer for each file that the program wants to read from, and one output buffer for each it wants to write to.

A stream can be in one of three states:

- **Unbuffered** Characters are intended to appear from the source or at the destination as soon as possible. Streams used for error reporting or logging might be unbuffered.

- **Fully buffered** Characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. Streams used for file I/O are normally fully buffered to optimize throughput.
- **Line buffered** Characters are intended to be transmitted to or from the host environment as a block when a newline character is encountered. Streams connected to interactive devices such as terminals are line-buffered when you open them.

In the next section, we'll introduce predefined streams and describe how they're buffered.

Predefined Streams

Your program has three *predefined text streams* open and available for use on startup. These predefined streams are declared in `<stdio.h>`:

```
extern FILE * stdin;  // standard input stream
extern FILE * stdout; // standard output stream
extern FILE * stderr; // standard error stream
```

The *standard output stream* (`stdout`) is the conventional output destination from the program. This stream is usually associated with the terminal that initiated the program but can be redirected to output to a file or other stream, as follows:

```
$ echo fred
fred
$ echo fred > tempfile
$ cat tempfile
fred
```

Here, the output from the `echo` command is redirected to *tempfile*.

The *standard input stream* (`stdin`) is the conventional input source for the program. By default, `stdin` is associated with the keyboard but may be redirected to input from a file, for example, with the following commands:

```
$ echo "one two three four five six seven" > fred
$ wc < fred
1 7 34
```

The contents of the file *fred* are redirected to the `stdin` of the `wc` command, which outputs the newline (1), word (7), and byte counts (34) from *fred*.

The *standard error stream* (`stderr`) is for writing diagnostic output. As initially opened, `stderr` isn't fully buffered; `stdin` and `stdout` are fully buffered if and only if the stream doesn't refer to an interactive device. The `stderr` stream isn't fully buffered, so that error messages can be viewed as soon as possible.

Figure 8-1 shows the predefined streams `stdin`, `stdout`, and `stderr` attached to the keyboard and display of the user's terminal.

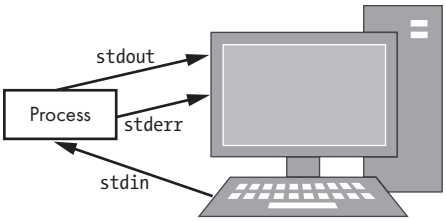

Figure 8-1: Standard streams attached to I/O communication channels

The output stream of one program can be redirected to be another application's input stream by using POSIX pipes. In many operating systems, you can chain applications together by separating commands with the vertical bar character (|):

```
$ echo "Hello Robert" | sed "s/Hello/Hi/" | sed "s/Robert/robot/"
Hi robot
```

Stream Orientation

Each stream has an *orientation* that indicates whether the stream contains narrow or wide characters. After a stream is associated with an external file, but before any operations are performed on it, the stream doesn't have an orientation. Once a wide character I/O function has been applied to a stream without orientation, the stream becomes a *wide-oriented stream*. Similarly, once a byte I/O function has been applied to a stream without orientation, the stream becomes a *byte-oriented stream*. Multibyte character sequences or narrow characters that can be represented as an object of type `char` (that are required by the C Standard to be 1 byte) can be written to a byte-oriented stream.

You can reset the orientation of a stream by using the `fwide` function or by closing and then reopening the file. Applying a byte I/O function to a wide-oriented stream, or a wide-character I/O function to a byte-oriented stream, results in undefined behavior. You should never mix narrow character data, wide character data, and binary data in the same file.

All three predefined streams (`stderr`, `stdin`, and `stdout`) are unoriented at program startup.

Text and Binary Streams

The C Standard supports both text streams and binary streams. A *text stream* is an ordered sequence of characters composed into lines, each of which consists of zero or more characters plus a terminating newline character sequence. You can denote a single line break on a Unix-like system by using a line feed (`\n`). Most Microsoft Windows programs use a carriage return (`\r`) followed by a line feed (`\n`).

The different newline conventions cause text files that have been transferred between systems with different conventions to display incorrectly. Text in files created with programs that are common on Unix-like

operating systems appear as a single, long line on older Microsoft Windows programs that don't display a single line feed or a single carriage return as a line break.

A *binary stream* is an ordered sequence of arbitrary binary data. Data read in from a binary stream will be the same as data written out earlier to that same stream, under the same implementation. These streams may, however, have an implementation-defined number of null bytes appended to the end of the stream.

Binary streams are always more capable and more predictable than a text stream. However, the easiest way to read or write an ordinary text file that can work with other text-oriented programs is through a text stream.

Opening and Creating Files

When you open or create a file, it's associated with a stream. The following functions open or create a file.

The fopen Function

The `fopen` function opens the file whose name is given as a string and pointed to by `filename`, and then associates a stream with it. If the file doesn't already exist, `fopen` will create it:

```
FILE *fopen(
  const char * restrict filename,
  const char * restrict mode
);
```

The `mode` argument points to one of the strings shown in Table 8-1 to determine how to open the file.

Table 8-1: Valid File Mode Strings

Mode string	Description
`r`	Open existing text file for reading
`w`	Truncate to zero length or create text file for writing
`a`	Append, open, or create text file for writing at end-of-file
`rb`	Open existing binary file for reading
`wb`	Truncate file to zero length or create binary file for writing
`ab`	Append, open, or create binary file for writing at end-of-file
`r+`	Open existing text file for reading and writing
`w+`	Truncate to zero length or create text file for reading and writing
`a+`	Append; open or create text file for update, writing at current end-of-file
`r+b` or `rb+`	Open existing binary file for update (reading and writing)
`w+b` or `wb+`	Truncate to zero length or create binary file for reading and writing
`a+b` or `ab+`	Append; open or create binary file for update, writing at current end-of-file

Opening a file with read mode (by passing `r` as the first character in the `mode` argument) fails if the file doesn't exist or cannot be read. Opening a file with append mode (by passing `a` as the first character in the `mode` argument) causes all subsequent writes to the file to occur at the current end of the file. In some implementations, opening a binary file with append mode (by passing `b` as the second or third character in the `mode` argument) may initially set the file position indicator for the stream beyond the last data written, because of null character padding.

You can open a file in update mode by passing `+` as the second or third character in the `mode` argument, allowing both read and write operations to be performed on the associated stream. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

The C11 Standard added the *exclusive mode* for reading and writing binary and text files, as shown in Table 8-2.

Table 8-2: Valid File Mode Strings Added by C11

Mode string	Description
`wx`	Create exclusive text file for writing
`wbx`	Create exclusive binary file for writing
`w+x`	Create exclusive text file for reading and writing
`w+bx` or `wb+x`	Create exclusive binary file for reading and writing

Opening a file with exclusive mode (by passing `x` as the last character in the `mode` argument) fails if the file already exists or cannot be created. Otherwise, the file is created with exclusive (also known as *nonshared*) access to the extent that the underlying system supports exclusive access.

On a final note, make sure that you never copy a `FILE` object. The following program, for example, can fail because a by-value copy of `stdout` is being used in the call to `fputs`:

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

int main(void) {
  FILE my_stdout = *stdout;
  if (fputs("Hello, World!\n", &my_stdout) == EOF) {
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

This program has undefined behavior and it typically results in a crash when executed.

The POSIX open Function

On POSIX systems, the `open` function (IEEE Std 1003.1:2018) establishes the connection between a file identified by `path` and a value called a *file descriptor*:

```
int open(const char *path, int oflag, ...);
```

The *file descriptor* is a non-negative integer that refers to the structure representing the file (called the *open file description*). The file descriptor returned by the `open` function is the smallest integer not yet returned by a prior call to `open` or passed to `close` and is unique to the calling process. The file descriptor is used by other I/O functions to refer to that file. The `open` function sets the file offset used to mark the current position within the file to the beginning of the file.

The value of the `oflag` parameter sets the open file description's *file access modes*, which specify whether the file is being opened for reading, writing, or both. Values for `oflag` are constructed by a bitwise-inclusive OR of a file access mode and any combination of access flags. Applications must specify exactly one of the following file access modes in the value of `oflag`:

`O_EXEC` Open for execute only (nondirectory files)

`O_RDONLY` Open for reading only

`O_RDWR` Open for reading and writing

`O_SEARCH` Open directory for search only

`O_WRONLY` Open for writing only

The value of the `oflag` parameter also sets the *file status flags*, which control the behavior of the `open` function and affect how file operations are performed. These flags include the following:

`O_APPEND` Sets the file offset to the end of the file prior to each write

`O_TRUNC` Truncates the length to 0

`O_CREAT` Creates a file

`O_EXCL` Causes the open to fail if `O_CREAT` is also set and the file exists

The `open` function takes a variable number of arguments. The value of the argument following the `oflag` argument specifies the file-mode bits (the file permissions when you create a new file) and is of type `mode_t`.

Listing 8-1 shows an example of using the `open` function to open a file for writing by the owner.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
//---snip---
int fd;
❶ mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
const char *pathname = "/tmp/file";
```

```
//---snip---
if ((fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)❷) == -1)
{
  fprintf(stderr, "Can't open %s.\n", pathname);
  exit(1);
}
//---snip---
```

Listing 8-1: Opening a file as write-only by the owner

We create a `mode` flag ❶ that is a bitwise-inclusive OR of the following mode bits for access permission:

`S_IRUSR` Read permission bit for the owner of the file

`S_IWUSR` Write permission bit for the owner of the file

`S_IRGRP` Read permission bit for the group owner of the file

`S_IROTH` Read permission bit for other users

The call to `open` ❷ takes multiple arguments, including the pathname of the file, the `oflag`, and the mode. The file access mode is `O_WRONLY`, which means the file is opened for writing only. The `O_CREAT` file status flag informs `open` to create the file; the `O_TRUNC` file status flag informs `open` that if the file exists and is successfully opened, it should discard the previous contents of the file but retain the identifier.

If the file was successfully opened, the `open` function returns a nonnegative integer representing the file descriptor. Otherwise, `open` returns –1 and sets `errno` to indicate the error. Listing 8-1 checks for a value of –1, writes a diagnostic message to the predefined `stderr` stream if an error occurred, and then exits.

In addition to `open`, POSIX has other useful functions for working with file descriptors, such as the `fileno` function to get the file descriptor associated with an existing file pointer, and the `fdopen` function to create a new stream file pointer from an existing file descriptor. POSIX APIs available through the file descriptor allow access to features of POSIX filesystems that aren't normally exposed through the file pointer interfaces such as directories, file permissions, and symbolic and hard links.

Closing Files

Opening a file allocates resources. If you continually open files without closing them, you'll eventually run out of file descriptors or handles available for your process, and attempting to open more files will fail. Consequently, it's important to close files after you've finished using them.

The fclose Function

The C Standard Library `fclose` function closes the file:

```
int fclose(FILE *stream);
```

Any unwritten buffered data for the stream is delivered to the host environment to be written to the file. Any unread buffered data is discarded.

It's possible for the `fclose` function to fail. When `fclose` writes the remaining buffered output, for example, it might return an error because the disk is full. Even if you know the buffer is empty, errors can still occur when closing a file if you're using the Network File System (NFS) protocol. Despite the possibility of failure, recovery is often impossible, so programmers commonly ignore errors returned by `fclose`. When closing the file fails, a common practice is to abort the process or to truncate the file so its contents are meaningful when next read.

The value of a pointer to a `FILE` object is indeterminate after the associated file is closed. Whether a file of zero length (in which an output stream hasn't written any data) actually exists is implementation defined.

You can reopen a closed file in the same program or another one, and its contents can be reclaimed or modified. If the `main` function returns, or if the `exit` function is called, all open files close (and all output streams are flushed) before program termination.

Other paths to program termination, such as calling the `abort` function, may not close all files properly, which means that buffered data not yet written to a disk might be lost.

The POSIX close Function

On POSIX systems, you can use the `close` function to deallocate the file descriptor specified by `fildes`:

```
int close(int fildes);
```

If an I/O error occurred while reading from or writing to the filesystem during `close`, it may return –1 with `errno` set to `EIO`; if this error is returned, the state of `fildes` is unspecified, meaning you can no longer read or write data to the descriptor or attempt to close it again.

Once a file is closed, the file descriptor no longer exists, because the integer corresponding to it no longer refers to a file. Files are also closed when the process owning that stream terminates.

An application that uses `fopen` to open a file must use `fclose` to close it; an application that uses `open` to open a file must use `close` to close it (unless it passed the descriptor to `fdopen`, in which case it must close by calling `fclose`).

Reading and Writing Characters and Lines

The C Standard defines functions for reading and writing specific characters or lines.

Most byte stream functions have counterparts that take a wide character (`wchar_t`) or wide character string instead of a narrow character (`char`)

or string, respectively (see Table 8-3). Byte stream functions are declared in the header file <stdio.h>, and the wide stream functions in <wchar.h>. The wide-character functions operate on the same streams (such as stdout).

Table 8-3: Narrow and Wide String I/O Functions

char	wchar_t	Description
fgetc	fgetwc	Reads a character from a stream
getc	getwc	Reads a character from a stream (often a macro)
getchar	getwchar	Reads a character from stdin
fgets	fgetws	Reads a line from a stream
fputc	fputwc	Writes a character to a stream
putc	putwc	Writes a character to a stream (often a macro)
fputs	fputws	Writes a string to a stream
putchar	putwchar	Writes a character to stdout
puts	N/A	Writes a string to stdout
ungetc	ungetwc	Returns a character to a stream
scanf	wscanf	Reads formatted character input from stdin
fscanf	fwscanf	Reads formatted character input from a stream
sscanf	swscanf	Reads formatted character input from a buffer
printf	wprintf	Prints formatted character output to stdout
fprintf	fwprintf	Prints formatted character output to a stream
sprintf	swprintf	Prints formatted character output to a buffer
snprintf	N/A	The same as sprintf with truncation. The swprintf function also takes a length argument, but behaves differently from snprintf in the way it interprets it.

In this chapter, we'll discuss the byte stream functions only. You may want to avoid wide-character function variants altogether and work exclusively with UTF-8 character encodings, if possible, as these functions are less prone to programmer error and security vulnerabilities.

The fputc function converts the character c to the type unsigned char and writes it to stream:

```
int fputc(int c, FILE *stream);
```

It returns EOF if a write error occurs; otherwise, it returns the character it has written.

The putc function is just like fputc, except that most implementations implement it as a macro:

```
int putc(int c, FILE *stream);
```

If putc is implemented as a macro, it may evaluate its arguments more than once, so the arguments should never be expressions with side effects.

Using fputc is generally safer. See CERT C rule FIO41-C (Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects) for more information.

The putchar function is equivalent to the putc function, except that it uses stdout as the value of the stream argument.

The fputs function writes the string s to the stream stream:

```
int fputs(const char * restrict s, FILE * restrict stream);
```

This function doesn't write the null character from the string s—nor does it write a newline character—but outputs only the characters in the string. If a write error occurs, fputs returns EOF. Otherwise, it returns a non-negative value. For example, the following statements output the text I am Groot, followed by a newline:

```
fputs("I ", stdout);
fputs("am ", stdout);
fputs("Groot\n", stdout);
```

The puts function writes the string s to the stream stdout followed by a newline:

```
int puts(const char *s);
```

The puts function is the most convenient function for printing simple messages because it takes only a single argument. Here's an example:

```
puts("This is a message.");
```

The fgetc function reads the next character as an unsigned char from a stream and returns its value, converted to an int:

```
int fgetc(FILE *stream);
```

If an end-of-file condition or read error occurs, the function returns EOF.

You may recall that the gets function reads characters from stdin and writes them into a character array until a newline or EOF is reached. The gets function is inherently insecure. It was deprecated in C99 and removed from C11 and *should never be used*. If you need to read a string from stdin, consider using the fgets function instead. The fgets function reads at most one less than the number of characters specified (leaving room for a null character) from a stream into a character array.

Stream Flushing

As described earlier in this chapter, streams can be fully or partially buffered, meaning that data you thought you wrote may not yet be delivered to the host environment. In particular, this can be a problem when the

program terminates abruptly. The `fflush` function delivers any unwritten data for a specified stream to the host environment to be written to the file:

```
int fflush(FILE *stream);
```

The behavior is undefined if the last operation on the stream was input. If the stream is a null pointer, the `fflush` function performs this flushing action on all streams. You should make sure that your file pointer isn't null before calling `fflush` if this isn't your intent. The `fflush` function sets the error indicator for the stream and returns `EOF` if a write error occurs; otherwise, it returns zero.

Setting the Position in a File

Random-access files (which include a disk file, for example, but not a terminal) maintain a file position indicator associated with the stream. The *file position indicator* describes where in the file the stream is currently reading or writing.

When you open a file, the indicator is positioned at the file's start. You can position the indicator whenever you wish to read or write any portion of the file. The `ftell` function obtains the current value of the file position indicator, while the `fseek` function sets the file position indicator. These functions use the `long int` type to represent offsets (positions) in a file and are therefore limited to offsets that can be represented as a `long int`. Listing 8-2 demonstrates the use of the `ftell` and `fseek` functions.

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

int main(void) {
  FILE *fp = fopen("fred.txt", "r");
  if (fp  == NULL) {
    fputs("Cannot open fred.txt file\n", stderr);
    return EXIT_FAILURE;
  }
  if (fseek(fp, 0, SEEK_END) != 0) {
    fputs("Seek to end of file failed\n", stderr);
    return EXIT_FAILURE;
  }
  long int fpi = ftell(fp);
  if (fpi == -1L) {
    perror("Tell");
    return EXIT_FAILURE;
  }
  printf("file position =  ld\n", fpi);
  if (fclose(fp) == EOF) {
    fputs("Failed to close file\n", stderr);
    return EXIT_FAILURE;
  }
```

```
  return EXIT_SUCCESS;
}
```

Listing 8-2: Using the `ftell` and `fseek` functions

This program opens a file called *fred.txt* and calls `fseek` to set the file position indicator to the end of the file (indicated by `SEEK_END`). The `ftell` function returns the current value of the file position indicator for the stream as a `long int`. The program prints out this value and exits. Finally, we close the file referenced by the `fp` file pointer. To ensure your code is robust, make sure you check for errors. File I/O, in particular, can fail for any number of reasons. The `fopen` function returns a null pointer when it fails. The `fseek` function returns nonzero only for a request that cannot be satisfied. On failure, the `ftell` function returns `-1L` and stores an implementation-defined value in `errno`. If the return value from `ftell` is equal to `-1L`, we use the `perror` function to print the string we provided "Tell" followed by a colon (:), a space, an appropriate error message corresponding to the value stored in errno, and finally, a newline character. The `fclose` function returns `EOF` if any errors were detected. One of the unfortunate aspects of the C Standard Library demonstrated by this short program is that each function tends to report errors in its own unique way, so you normally need to refer to your documentation to see how to test for errors.

The newer `fgetpos` and `fsetpos` functions use the `fpos_t` type to represent offsets. This type can represent arbitrarily large offsets, meaning you can use `fgetpos` and `fsetpos` with arbitrarily large files. A wide-oriented stream has an associated `mbstate_t` object that stores the stream's current parse state. A successful call to `fgetpos` stores this multibyte state information as part of the value of the `fpos_t` object. A later successful call to `fsetpos` using the same stored `fpos_t` value restores the parse state as well as the position within the controlled stream. It's not possible to convert an `fpos_t` object to an integer byte or character offset within the stream except indirectly by calling `fsetpos` followed by `ftell`. The short program shown in Listing 8-3 demonstrates the use of the `fgetpos` and `fsetpos` functions.

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

int main(void) {
  FILE *fp = fopen("fred.txt", "w+");
  if (fp  == NULL) {
    fputs("Cannot open fred.txt file\n", stderr);
    return EXIT_FAILURE;
  }
  fpos_t pos;
  if (fgetpos(fp, &pos) != 0) {
    perror("get position");
    return EXIT_FAILURE;
  }
  if (fputs("abcdefghijklmnopqrstuvwxyz", fp) == EOF) {
      fputs("Cannot write to fred.txt file\n", stderr);
  }
```

```
  if (fsetpos(fp, &pos) != 0) {
    perror("set position");
    return EXIT_FAILURE;
  }
  long int fpi = ftell(fp);
  if (fpi == -1L) {
    perror("seek");
    return EXIT_FAILURE;
  }
  printf("file position = %ld\n", fpi);
  if (fputs("0123456789", fp) == EOF) {
    fputs("Cannot write to fred.txt file\n", stderr);
  }
  if (fclose(fp) == EOF) {
    fputs("Failed to close file\n", stderr);
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Listing 8-3: Using the `fgetpos` and `fsetpos` functions

This program opens the *fred.txt* file for writing and then calls `fgetpos` to get the current file position within the file, which is stored in `pos`. We then write some text to the file before calling `fsetpos` to restore the file position indicator to the position stored in `pos`. At this point, we can use the `ftell` function to retrieve and print the file position, which should be 0. After running this program, *fred.txt* contains the following text:

```
0123456789klmnopqrstuvwxyz
```

You cannot write to a stream and then read from it again without an intervening call to the `fflush` function to write any unwritten data or to a file positioning function (`fseek`, `fsetpos`, or `rewind`). You also cannot read from a stream and then write to it without an intervening call to a file positioning function.

The `rewind` function sets the file position indicator to the beginning of the file:

```
void rewind(FILE *stream);
```

The `rewind` function is equivalent to the invoking `fseek` followed by `clearerr` to clear the error indicator for the stream:

```
fseek(stream, 0L, SEEK_SET);
clearerr(stream);
```

You shouldn't attempt to use file positions in files opened in append mode, because many systems don't modify the current file position indicator for append or will forcefully reset to the end of the file when writing. If using the APIs that use file positions, the file position indicator is

maintained by subsequent reads, writes, and positioning requests. Both POSIX and Windows have APIs that never use the file position indicator; for those, you always specify the offset into the file at which to perform I/O.

Removing and Renaming Files

The C Standard Library provides a `remove` function to delete a file, and a `rename` function to move or rename it:

```
int remove(const char *filename);
int rename(const char *old, const char *new);
```

In POSIX, the file deletion function is named `unlink`:

```
int unlink(const char *path);
```

The `unlink` function has better-defined semantics because it's specific to POSIX filesystems. POSIX also uses `rename` for renaming. In POSIX and Windows, we can have any number of links to a file, including hard links and open file descriptors. The `unlink` function always removes the directory entry for the file, but it deletes the actual file only when there are no more links to it. Even at this point, the actual contents of the file may remain in permanent storage.

The `remove` function behaves the same as the `unlink` function on POSIX systems but may have different behavior on other operating systems.

Using Temporary Files

We frequently use *temporary files* as an interprocess communication mechanism or for temporarily storing information out to disk to free up RAM. For example, one process might write to a temporary file that another process reads from. These files are normally created in a temporary directory by using functions such as the C Standard Library's `tmpfile` and `tmpnam` or POSIX's `mkstemp`.

Temporary directories can be either global or user specific. In Unix and Linux, the `TMPDIR` environment variable is used to specify the location of the global temporary directories, which are typically */tmp* and */var/tmp*. Linux usually has user-specific temporary directories defined by the `$XDG_RUNTIME_DIR` environment variable, which is typically set to */run/user/$uid*. In Windows, you can find user-specific temporary directories in the *AppData* section of the User Profile, typically *C:\Users\User Name\AppData\Local\Temp* (*%USERPROFILE%\AppData\Local\Temp*). On Windows, the global temporary directory is specified by either the `TMP`, `TEMP`, or `USERPROFILE` environment variable. The *C:\Windows\Temp* directory is a system folder used by Windows to store temporary files.

For security reasons, it's best for each user to have their own temporary directory, because the use of global temporary directories frequently results

in security vulnerabilities. The most secure function for creating temporary files is the POSIX `mkstemp` function. However, because accessing files in shared directories may be difficult or impossible to implement securely, we recommended that you not use any of the available functions, and instead perform the interprocess communication by using sockets, shared memory, or other mechanisms designed for this purpose.

Reading Formatted Text Streams

In this section, we'll demonstrate the use of the `fscanf` function to read formatted input. The `fscanf` function is the corresponding input version of the `fprintf` function that we introduced all the way back in Chapter 1 and has the following signature:

```
int fscanf(FILE * restrict stream, const char * restrict format, ...);
```

The `fscanf` function reads input from the stream pointed to by `stream`, under control of the `format` string that tells the function how many arguments to expect, their type, and how to convert them for assignment. Subsequent arguments are pointers to the objects receiving the converted input. If you provide fewer arguments than conversion specifiers, the behavior is undefined. If you provide more arguments than conversion specifiers, the excess arguments are evaluated but otherwise ignored. The `fscanf` function has a great deal of functionality that we'll only touch upon here. For more information, refer to the C Standard.

To demonstrate the use of `fscanf`, as well as some other I/O functions, we'll implement a program that reads in the *signals.txt* file shown in Listing 8-4 and print each line. Each line of this file contains the following:

- A signal number (a small, positive integer value)
- The signal ID (a small string of up to six alphanumeric characters)
- A short string with a description of the signal

Fields are whitespace delimited except for the description, which can contain whitespace and is delimited by a newline:

```
1 HUP   Hangup
2 INT   Interrupt
3 QUIT Quit
4 ILL Illegal instruction
5 TRAP Trace trap
6 ABRT Abort
7 EMT EMT trap
8 FPE Floating-point exception
```

Listing 8-4: The signals.txt *file*

Listing 8-5 shows the *signals* program, which reads this file and prints out each line.

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

int main(void) {
  int status = EXIT_SUCCESS;
  FILE *in;

  struct sigrecord {
      int signum;
      char signame[10];
      char sigdesc[100];
❶ } sigrec;

  if ((in = fopen("signals.txt", "r")) == NULL) {
    fputs("Cannot open signals.txt file\n", stderr);
    return EXIT_FAILURE;
  }

  do {
❷   int n = fscanf(in, "%d%9s%*[ \t]%99[^\n]",
      &sigrec.signum, sigrec.signame, sigrec.sigdesc
    );
    if (n == 3) {
      printf(
        "Signal\n  number = %d\n  name = %s\n  description = %s\n\n",
        sigrec.signum, sigrec.signame, sigrec.sigdesc
      );
    }
    else if (n != EOF) {
      fputs("Failed to match signum, signame or sigdesc\n", stderr);
      status = EXIT_FAILURE;
      break;
    }
    else {
      break;
    }
❸ } while (1);

❹ if (fclose(in) == EOF) {
    fputs("Failed to close file\n", stderr);
    status = EXIT_FAILURE;
  }
  return status;
}
```

Listing 8-5: The signals *program*

We define several variables in the `main` function, including the `sigrec` structure ❶, which we'll use to store the signal information found on each line of the file. The `sigrec` structure contains three members: a `signum` member of

type `int` that will hold the signal number; a `signame` member that's an array of `char` and will hold the signal ID; and the `sigdesc` member, an array of `char` that will hold the description of the signal. Both arrays have fixed sizes that we determined were adequately sized for the strings being read from the file. If the strings read from the file are too long to fit in these arrays, the program will treat it as an error.

The call to `fscanf` ❷ reads each line of input from the file. It appears inside of an infinite `do...while (1)` loop ❸ that we must break out for the program to terminate. We assign the return value from the `fscanf` function to a local variable `n`. The `fscanf` function returns `EOF` if an input failure occurs before the first conversion has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure. The call to `fscanf` assigns three input items, so we only print the signal description when `n` is equal to 3. Otherwise, if `n` does not equal `EOF`, a matching failure occurred, so we output an appropriate diagnostic message to `stderr`, set `status` to `EXIT_FAILURE`, and break out of the loop. The final possibility is that `fscanf` returns `EOF` indicating that the end of the file was reached, in which case we simply break out of the loop without altering `status`.

The `fscanf` function uses a *format string* to inform it how to assign the input text to each argument. In this case, the `"%d%9s%*[ \t]%99[^\n]"` format string contains four *conversion specifiers*, which specify how the input read from the stream is converted into values stored in the objects referenced by the format string's arguments. We introduce each conversion specification with the character `%`. After the `%`, the following may appear, in sequence:

- An optional character `*` that discards the input without assigning it to an argument
- An optional integer greater than zero that specifies the maximum field width (in characters)
- An optional length modifier that specifies the size of the object
- A conversion specifier character that specifies the type of conversion to be applied

The first conversion specifier in the format string is `%d`. This conversion specifier matches the first optionally signed decimal integer—which should correspond to the signal number in the file—and stores the value in the third argument referenced by `sigrec.signum`. Without an optional length modifier, the length of the input depends on the conversion specifier's default type. For the `d` conversion specifier, the argument must point to a `signed int`.

The second conversion specifier in this format string is `%9s`, which matches the next sequence of nonwhitespace characters from the input stream—corresponding to the signal name—and stores these characters as a string in the fourth argument referenced by `sigrec.signame`. The length modifier prevents more than nine characters from being input then writes a null character in `sigrec.signame` after the characters that have been matched. A conversion specifier of `%10s` in this example would allow a buffer

overflow to occur. A conversion specifier of `%9s` can still fail to read the entire string, resulting in a matching error. When reading data into a fixed-size buffer as we are doing, it is a good idea to test inputs that exactly match or slightly exceed the fixed buffer length to ensure buffer overflow does not occur and that the string is properly null-terminated.

We're going to skip the third conversion specifier for a moment and talk about the fourth one: `%99[^\n]`. This fancy conversion specifier will match the signal description field in the file. The brackets ([]) contain a *scanset*, which is similar to a regular expression. This scanset uses the circumflex (^) to exclude `\n` characters. Put together, `%99[^\n]` reads all the characters until it reaches a `\n` (or `EOF`), and stores them in the fifth argument referenced by `sigrec.sigdesc`. C programmers commonly use this syntax to read an entire line. This conversion specifier also includes a maximum string length of 99 characters to avoid buffer overflows.

Finally, we'll circle back to the third conversion specifier: `%*[ \t]`. As we have just seen, the fourth conversion specifier reads all the characters, starting from the end of the signal ID. Unfortunately, this includes any whitespace characters between the signal ID and the start of the description. The purpose of the `%*[ \t]` conversion specifier is to consume any space or horizontal tab characters between these two fields and suppress them by using the assignment-suppressing specifier *. It's also possible to include other whitespace characters in the scanset for this conversion specifier.

Finally, we close the file by calling the `fclose` function ❹.

Reading to and Writing from Binary Streams

The C Standard Library `fread` and `fwrite` functions operate on a binary stream. The `fwrite` function has the following signature:

```
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
  FILE * restrict stream);
```

This function writes up to `nmemb` elements of `size` bytes from the array pointed to by `ptr` to `stream`. It does so by converting each object to an array of `unsigned char` (every object can be converted to an array of this type), and then calling the `fputc` function to write the value of each character in the array in order. The file position indicator for the stream is advanced by the number of characters successfully written.

Listing 8-6 demonstrates the use of the `fwrite` function to write signal records to the *signals.txt* file.

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

typedef struct sigrecord {
  int signum;
  char signame[10];
```

```
  char sigdesc[100];
} sigrecord;

int main(void) {
  int status = EXIT_SUCCESS;
  FILE *fp;
  sigrecord sigrec;

❶ if ((fp = fopen("signals.txt", "wb")) == NULL) {
    fputs("Cannot open signals.txt file\n", stderr);
    return EXIT_FAILURE;
  }

❷ sigrecord sigrec30 = { 30, "USR1", "user-defined signal 1" };
  sigrecord sigrec31 = {
    .signum = 31, .signame = "USR2", .sigdesc = "user-defined signal 2"
  };

  size_t size = sizeof(sigrecord);

❸ if (fwrite(&sigrec30, size, 1, fp) != 1) {
    fputs("Cannot write sigrec30 to signals.txt file\n", stderr);
    status = EXIT_FAILURE;
    goto close_files;
  }

  if (fwrite(&sigrec31, size, 1, fp) != 1) {
    fputs("Cannot write sigrec31 to signals.txt file\n", stderr);
    status = EXIT_FAILURE;
  }

close_files:
  if (fclose(fp) == EOF) {
    fputs("Failed to close file\n", stderr);
    status = EXIT_FAILURE;
  }

  return status;
}
```

Listing 8-6: Writing to a binary file using direct I/O

We open the *signals.txt* file in `wb` mode ❶ to create a binary file for writing. We declare two `sigrecord` structures ❷ and initialize them with the signal values we want to write to the file. The second structure, `sigrec31`, is initialized using designated initializers for comparison. Both initialization styles have the same behavior; designated initializers make the declaration less terse but clearer. The actual writing begins at ❸. We check the return values from each call to the `fwrite` function to ensure that it wrote the correct number of elements.

Listing 8-7 uses the `fread` function to read the data we just wrote from the *signals.txt* file.

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

typedef struct sigrecord {
  int signum;
  char signame[10];
  char sigdesc[100];
} sigrecord;

int main(void) {
  int status = EXIT_SUCCESS;
  FILE *fp;
  sigrecord sigrec;
  size_t size = sizeof(sigrecord);

❶ if ((fp = fopen("signals.txt", "rb")) == NULL) {
    fputs("Cannot open signals.txt file\n", stderr);
    return EXIT_FAILURE;
  }

  // read the second signal
❷ if (fseek(fp, size, SEEK_SET) != 0) {
    fputs("fseek in signals.txt file failed\n", stderr);
    status = EXIT_FAILURE;
    goto close_files;
  }

❸ if (fread(&sigrec, size, 1, fp) != 1) {
    fputs("Cannot read from signals.txt file\n", stderr);
    status = EXIT_FAILURE;
    goto close_files;
  }

  printf(
    "Signal\n  number = %d\n  name = %s\n  description = %s\n\n",
    sigrec.signum, sigrec.signame, sigrec.sigdesc
  );

close_files:
  fclose(fp);
  return status;
}
```

Listing 8-7: Reading from a binary file using direct I/O

We open the binary file by using the `rb` mode ❶ for reading. Next, to make this example a bit more interesting, the program reads and prints the information for a specific signal, rather than reading the entire file. We could indicate which signal to read by using an argument to the program, but for this example, we hardcoded it as the second signal. To accomplish this, the program invokes the `fseek` function ❷ to set the file position indicator for the stream referenced by `fp`. As you learned earlier in this chapter,

the file position indicator determines the file position for the subsequent I/O operation. For a binary stream, we set the new position by adding the offset (measured in bytes) to the position specified by the final argument (the beginning of the file, as indicated by `SEEK_SET`). The first signal is at position 0 in the file, and each subsequent signal is at an integer multiple of the size of the structure from the beginning of the file.

After the file position indicator is positioned at the start of the second signal, we call the `fread` function ❸ to read the data from the binary file into the structure referenced by `&sigrec`. Similar to `fwrite`, the `fread` function reads up to one element, whose size is specified by `size`, from the stream pointed to by `fp`. In most cases, this object has the size and type of the corresponding call to `fwrite`. The file position indicator for the stream is advanced by the number of characters successfully read. We check the return value from the `fread` function to ensure the correct number of elements was read.

Binary files can have different formats. In particular, the order of bytes within a binary representation of a number, or endianness, can vary between systems. A *big-endian ordering* places the most significant byte first and the least significant byte last, while a *little-endian ordering* does the opposite. For example, consider the unsigned hexadecimal number `0x1234`, which requires at least two bytes to represent. In a big-endian ordering, these two bytes are `0x12`, `0x34`, while in a little-endian ordering, the bytes are arranged as `0x34`, `0x12`. Intel and AMD processors use the little-endian format, while the ARM and POWER series of processors can switch between either little- or big-endian format. However, big-endianness is the dominant ordering in network protocols such as IP, TCP, and UDP. Endianness is a problem when a binary file is created on one computer and is read on another computer with different endianness. Endianness independence in binary data formats can be achieved by always storing the data in one fixed endianness, or including a field in the binary file to indicate the endianness of the data.

Summary

In this chapter, you learned about I/O as well as C Standard streams, including stream buffering, the predefined streams, stream orientation, and the difference between text and binary streams.

You then learned how to create, open, and close files by using the C Standard Library and POSIX APIs. You also learned how to read and write characters and lines, read and write formatted text, and read and write from binary streams. You looked at how to flush a stream, set the position in a file, remove and rename files. Finally, you learned about temporary files and to avoid using them.

In the next chapter, you'll learn about compilation process and the preprocessor, including file inclusion, conditional inclusion, and macros.

9

PREPROCESSOR

with Aaron Ballman

The *preprocessor* is the part of the C compiler that runs at an early phase of compilation and transforms the source code before it's translated, such as inserting code from one file (typically a header file) into another (typically a source file). The preprocessor also allows you to specify that an identifier should be automatically substituted by a source code segment during macro expansion. In this chapter, you'll learn how to use the preprocessor to include files, define object- and function-like macros, and conditionally include code based on implementation-specific features.

The Compilation Process

Let's first see where the preprocessor sits in the compilation process. Conceptually, the compilation process consists of a pipeline of eight phases, as shown in Figure 9-1. We call these *translation phases* because each phase translates the code for processing by the next phase.

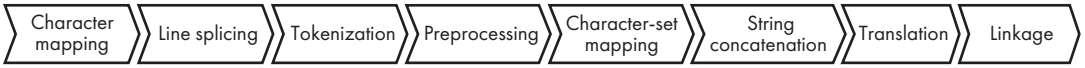

Figure 9-1: Translation phases

The preprocessor runs before the source code is translated into object code by the translator. This allows the preprocessor to modify the source code written by the user *before* it's operated on by the translator. Consequently, the preprocessor has a limited amount of semantic information about the program being compiled. It doesn't know about functions, variables, or types. Instead, only basic elements, such as header names, identifiers, literals, and punctuation characters like +, -, and ! are meaningful to the preprocessor. These basic elements, called *tokens*, are the smallest elements of a computer program that have meaning to a compiler.

The preprocessor operates on *preprocessing directives* that you include in the source code to program the behavior of the preprocessor. You spell preprocessing directives with a leading # token followed by a directive name, such as `#include`, `#define`, or `#if`. You can include whitespace between the beginning of the line and the # or between the # and the directive to indent directives. Terminate each preprocessing directive with a newline character.

A preprocessing directive causes the preprocessor to take an action that might alter the resulting translation unit, meaning the code you write is often not the same code consumed by the translator. Compiler implementations usually provide a way to view the preprocessor output, called a *translation unit*, passed to the translator. Viewing the preprocessor output is unnecessary, but you may find it informative to see the actual code given to the translator. Table 9-1 lists flags used by common compilers to output a translation unit. Preprocessed output files are commonly given a *.i* file extension.

Table 9-1: Outputting a Translation Unit

Compiler	Example command line
Clang	`clang` *`other-options`* `-E -o` *`output_file.i source.c`*
GCC	`gcc` *`other-options`* `-E -o` *`output_file.i source.c`*
Visual C++	`cl` *`other-options`* `/P /Fi`*`output_file.i source.c`*

File Inclusion

A powerful feature of the preprocessor is the ability to insert the contents of one source file into the contents of another source file by using the `#include` preprocessing directive. The included files are called *header files* to distinguish them from other *source files*. Header files typically contain declarations for use by other programs. This is the most common way to share external declarations of functions and objects with other parts of the program. You've already seen many examples of including the headers for C Standard Library functions in the examples in this book. For instance, the program in Table 9-2 is separated into a header file named *bar.h* and a source file named *foo.c*. The source file *foo.c* does not directly contain a declaration of `func`, yet the function is successfully referenced by name within `main`. During preprocessing, the `#include` directive inserts the contents of *bar.h* into *foo.c* in the place of the `#include` directive itself.

Table 9-2: Header File Inclusion

Original sources	Resulting translation unit

bar.h
`int func(void);`
	`int func(void);`
	
	`int main(void) {`
	`  return func();`
	`}`

foo.c
`#include "bar.h"`
 	
`int main(void) {`
`  return func();`
`}`	

The preprocessor executes `#include` directives as it encounters them. Therefore, inclusion has transitive properties: if a source file includes a header file that itself includes another header file, the preprocessed output will contain the contents of both header files. For example, given the *baz.h* and *bar.h* header files and the *foo.c* source file, the output after running the preprocessor on the *foo.c* source code is shown in Table 9-3.

Table 9-3: Transitive Header File Inclusion

Original sources	Resulting translation unit

baz.h
`int other_func(void);`
	`int other_func(void);`
	
	`int func(void);`
	
	`int main(void) {`
	`  return func();`
	`}`

bar.h
`#include "baz.h"`
 	
`int func(void);`	

foo.c
`#include "bar.h"`
 	
`int main(void) {`
`  return func();`
`}`	

Compiling the *foo.c* source file causes the preprocessor to include the `"bar.h"` header file. The preprocessor then finds the include directive for the `"baz.h"` header file and includes it as well, bringing the declaration for `other_func` into the resulting translation unit.

Quoted and Angle Bracket Include Strings

You can use either a quoted include string (for example, `#include "foo.h"`) or an angle-bracketed include string (for example, `#include <foo.h>`) to specify the file to include. The difference between these syntaxes is implementation defined, but typically influences the search path used to find the included files. For example, both Clang and GCC attempt to find files included with:

- Angle brackets on the *system include path*, specified using the `-isystem` flag
- Quoted strings on the *quoted include path*, specified using the `-iquote` flag

Refer to your compiler's documentation for the specific differences between these two syntaxes. Normally, headers for standard or system libraries are found on the default system include path, and your own project headers are found on the quoted include path.

Conditional Inclusion

Frequently, you'll need to write different code to support different implementations. For example, you may want to provide alternative implementations of a function for different target architectures. One solution to this problem is to maintain two files with slight variations between them and compile the appropriate file for a particular implementation. A better solution is to either translate or refrain from translating the target-specific code based on a preprocessor definition.

You can conditionally include source code by including a predicate condition using the preprocessing directives `#if`, `#elif`, and `#else`. A *predicate condition* is the controlling constant expression that's evaluated to determine which branch of the program the preprocessor should take. They're typically used along with the preprocessor `defined` operator, which determines if a given identifier is the name of a defined macro.

The conditional inclusion directives are similar to the `if` and `else` statements. When the predicate condition evaluates to a nonzero preprocessor value, the `#if` branch is processed, and all other branches are not. When the predicate condition evaluates to zero, the next `#elif` branch, if any, has its predicate tested for inclusion. If none of the predicate conditions evaluate to nonzero, then the `#else` branch, if there is one, is processed. The `#endif` preprocessing directive indicates the end of the conditionally included code.

The `defined` operator evaluates to `1` if the given identifier is defined as a macro, or `0` otherwise. For example, the preprocessing directives shown in Listing 9-1 conditionally determine which header file contents to include in the resulting translation unit. The preprocessed output of *foo.c* depends on whether `_WIN32` or `__ANDROID__` is a defined macro. If neither is a defined macro, the preprocessor output will be empty.

```
/* foo.c */
#if defined(_WIN32)
#include <Windows.h>
#elif defined(__ANDROID__)
#include <android/log.h>
#endif
```

Listing 9-1: Conditional inclusion

Unlike with the `if` and `else` keywords, preprocessor conditional inclusion cannot use braces to denote the block of statements controlled by the predicate. Instead, preprocessor conditional inclusion will include all the tokens from the `#if`, `#elif`, or `#else` directive (following the predicate) to the next balanced `#elif`, `#else`, or `#endif` token found, while skipping any tokens in a conditional inclusion branch not taken. Conditional inclusion directives can be nested.

You can write `#ifdef` *`identifier`* as shorthand for `#if defined` *`identifier`* or, equivalently, `#if defined(`*`identifier`*`)`. The parentheses around the identifier are optional. Similarly, you can write `#ifndef` *`identifier`* as shorthand for `#if !defined` *`identifier`*. There is no shorthand for `#elif defined` *`identifier`* or `#elif !defined` *`identifier`*.

Generating Errors

A conditional inclusion directive may need to generate an error if the preprocessor can't take any of the conditional branches because no reasonable fallback behavior exists. Consider the example in Listing 9-2, which uses conditional inclusion to select between including the C Standard Library header `<threads.h>` or the POSIX threading library header `<pthread.h>`. If neither option is available, you should alert the programmer porting the system that the code must be repaired.

```
#if __STDC__ && __STDC_NO_THREADS__ != 1
#include <threads.h>
//---snip---
#elif POSIX_THREADS == 200809L
#include <pthread.h>
//---snip---
#else
int compile_error[-1]; // Induce a compilation error
#endif
```

Listing 9-2: Inducing a compilation error

Here, the code generates a diagnostic but doesn't describe the actual problem. For this reason, C has the `#error` preprocessing directive, which causes the implementation to produce a diagnostic message. You can optionally follow this directive with one or more preprocessor tokens to include in the resulting diagnostic message. Using these, we can replace the erroneous array declaration from Listing 9-2 with an `#error` directive such as the one shown in Listing 9-3.

```
#if __STDC__ && __STDC_NO_THREADS__ != 1
#include <threads.h>
//---snip---
#elif POSIX_THREADS == 200809L
#include <pthread.h>
//---snip---
#else
#error Neither <threads.h> nor <pthread.h> is available
#endif
```

Listing 9-3: #error directive

This code generates the following error message if neither threading library header is available:

```
Neither <threads.h> nor <pthread.h> is available
```

Using Header Guards

One problem you will face when writing header files is preventing programmers from including the same file twice in a translation unit. Given that you can transitively include header files, you could easily include the same header file multiple times by accident (possibly even leading to infinite recursion between header files).

Header guards ensure that a header file is included only once per translation unit. A header guard is a design pattern that conditionally includes the contents of a header file based on whether a header-specific macro is defined. If the macro is not already defined, you define it so that a subsequent test of the header guard will not conditionally include the code. In the program shown in Table 9-4, *bar.h* uses a header guard (shown in bold font) to prevent its (accidental) duplicate inclusion from *foo.c*.

The first time that `"bar.h"` is included, the test to see that `BAR_H` is not defined will return `true`. We then define the macro `BAR_H` with an empty replacement list, which is sufficient to define `BAR_H`, and the function definition for `func` is included. The second time that `"bar.h"` is included, the preprocessor will not generate any tokens, because the conditional inclusion test will return `false`. Consequently, `func` is defined only once in the resulting translation unit.

Table 9-4: Header Guard

Original sources	Resulting translation unit

bar.h
`#ifndef BAR_H`
`#define BAR_H`
`int func(void) { return 1; }`
`#endif /* BAR_H */`
	`int func(void) { return 1; }`
	
	`int main(void) {`
	`  return func();`
	`}`

foo.c
`#include "bar.h"`
`#include "bar.h" // Repeated inclusion is`
`// usually not this obvious.`
 	
`int main(void) {`
`  return func();`
`}`	

A common practice when picking the identifier to use as a header file guard is to use the salient parts of the file path, filename, and extension, separated by an underscore and written in all capital letters. For example, if you had a header file that would be included with `#include "foo/bar/baz.h"`, you might choose `FOO_BAR_BAZ_H` as the header guard identifier.

Some IDEs will automatically generate the header guard for you. Avoid using a reserved identifier as your header guard's identifier, which could introduce undefined behavior. Identifiers that begin with an underscore followed by a capital letter are reserved. For example, `_FOO_H` is a reserved identifier and a bad choice for a user-chosen header guard identifier, even if you're including a file named *_foo.h*. Using a reserved identifier can result in a collision with a macro defined by the implementation, leading to a compilation error or incorrect code.

Macro Definitions

The `#define` preprocessing directive defines a macro. *Macros* can be used to define constant values or function-like constructs with generic parameters. The macro definition contains a (possibly empty) *replacement list*—a code pattern that's injected into the translation unit when the preprocessor expands the macro:

```
#define identifier replacement-list
```

The `#define` preprocessing directive is terminated with a newline. In the following example, the replacement list for `ARRAY_SIZE` is `100`.

```
#define ARRAY_SIZE 100
int array[ARRAY_SIZE];
```

In this example, the `ARRAY_SIZE` identifier is replaced by `100`. If no replacement list is specified, the preprocessor will simply remove the macro name. You can typically specify a macro definition on your compiler's command line, for example, using the `-D` flag in Clang and GCC or the `/D` flag in Visual C++. For Clang and GCC, the command line option `-DARRAY_SIZE=100` specifies that the macro identifier `ARRAY_SIZE` is replaced by `100`, producing the same result as the `#define` preprocessing directive from the previous example. If you do not specify the macro replacement list on the command line, compilers will typically provide a replacement list. For example, `-DFOO` is typically identical to `#define FOO 1`.

The scope of a macro lasts until the preprocessor encounters either an `#undef` preprocessing directive specifying that macro or the end of the translation unit. Unlike variable or function declarations, a macro's scope is independent of any block structure.

You can use the `#define` directive to define either an object-like macro or a function-like macro. A *function-like* macro is parameterized and requires passing a (possibly empty) set of arguments when you invoke it, similar to the way you would invoke a function. Unlike functions, macros let you perform operations using the symbols in the source file. That means you can create a new variable name, or reference the source file and line number the macro is on. An *object-like* macro is a simple identifier that will be replaced by a code fragment.

Table 9-5 illustrates the difference between function-like and object-like macros. `FOO` is an object-like macro that is replaced by the tokens `(1 + 1)` during macro expansion, and `BAR` is a function-like macro that is replaced by the tokens `(1 + (x))`, where `x` is whatever parameter is specified when invoking `BAR`.

Table 9-5: Macro Definition

Original source	Resulting translation unit

`#define FOO (1 + 1)`
`#define BAR(x) (1 + (x))`
 	
`int i = FOO;`
`int j = BAR(10);`
`int k = BAR(2 + 2);`
	`int i = (1 + 1);`
	`int j = (1 + (10));`
	`int k = (1 + (2 + 2));`

The opening parenthesis of a function-like macro definition must immediately follow the macro name, with no intervening whitespace. If a space appears between the macro name and the opening parenthesis, the

parenthesis simply becomes part of the replacement list, as is the case with the object-like `FOO` macro. The macro replacement list terminates with the first newline character in the macro definition. However, you can join multiple source lines by using the backslash (`\`) character followed by a newline to make your macro definitions easier to understand. For example, the following definition of the `cbrt` type-generic macro that computes the cube root of its floating argument

```
#define cbrt(X) _Generic((X), \
  long double: cbrtl(X), \
  default: cbrt(X), \
  float: cbrtf(X) \
)
```

is equivalent to, but easier to read, than the following:

```
#define cbrt(X) _Generic((X), long double: cbrtl(X), default: cbrt(X), float: cbrtf(X))
```

One danger when defining a macro is you can no longer use the macro's identifier in the rest of the program without inducing a macro replacement. For example, as a result of macro expansion, the following invalid program will not compile:

```
#define foo (1 + 1)
void foo(int i);
```

This is because the declaration of `foo` that the translator receives from the preprocessor is the following invalid declaration:

```
void (1 + 1)(int i);
```

You can solve this problem by consistently adhering to an idiom throughout your program, such as defining macro names with all uppercase letters, or prefixing all macro names with a mnemonic, as done in some styles of Hungarian notation.[12]

After you've defined a macro, the only way to redefine it is to first invoke the `#undef` directive for the macro. Once you've undefined it, the named identifier no longer represents a macro. For example, the program shown in Table 9-6 defines a function-like macro, includes a header file that uses the macro, and then undefines the macro so that it can be redefined later.

12. *Hungarian notation* is an identifier-naming convention in which the name of a variable or function indicates its intention or kind, and in some dialects, its type.

Table 9-6: Undefining Macros

Original sources	Resulting translation unit

header.h
`NAME(first)`
`NAME(second)`
`NAME(third)`
	`enum Names {`
	`  first,`
	`  second,`
	`  third,`
	`};`
	
	`void func(enum Names Name) {`
	`  switch (Name) {`
	`  case first:`
	`  case second:`
	`  case third:`
	`  }`
	`}`

file.c
`enum Names {`
`#define NAME(X) X,`
`#include "header.h"`
`#undef NAME`
`};`
 	
`void func(enum Names Name) {`
`  switch (Name) {`
`#define NAME(X) case X:`
`#include "header.h"`
`#undef NAME`
`  }`
`}`	

The first use of the `NAME` macro declares the names of enumerators within the `Names` enumeration. The `NAME` macro is undefined and then redefined to generate the `case` labels in a `switch` statement.

A common idiom is to undefine a macro before redefining it, as shown in Listing 9-4.

```
#undef NAME
#define NAME(X) X
```

Listing 9-4: Idiom for safely defining macros

Undefining a macro is safe even when the named identifier isn't the name of a macro. This macro definition works regardless of whether `NAME` is already defined.

Macro Replacement

Function-like macros may look like functions, but have different behavior. For example, macros let you perform operations using the symbols in the source file. With macros, unlike with functions, you can create new variable names or reference the source file and line number of the macro. When the preprocessor encounters a macro identifier, it will invoke the macro, which expands the identifier to replace it with the tokens from the replacement list, if any, specified in the macro's definition.

For function-like macros, the preprocessor replaces all parameters in the replacement list with the corresponding arguments in the macro invocation after expanding them. Any parameter in the replacement list preceded by a `#` token is replaced with a string literal preprocessing token

that contains the text of the argument preprocessing tokens (a process sometimes called *stringizing*). The `SRINGIZE` macro in Table 9-7 stringizes the value of `x`.

Table 9-7: Stringizing

Original source	Resulting translation unit
`#define STRINGIZE(x) #x` `const char *str = STRINGIZE(12);`	`const char *str = "12";`

The preprocessor also deletes all instances of the ## preprocessing token in the replacement list, concatenating the preceding preprocessing token with the following token, which is called *token pasting*. The `PASTE` macro in Table 9-8 is used to create a new identifier by concatenating `foo`, the underscore character (`_`), and `bar`.

Table 9-8: Token Pasting

Original source	Resulting translation unit
`#define PASTE(x, y) x ## _ ## y` `int PASTE(foo, bar) = 12;`	`int foo_bar = 12;`

After expanding the macro, the preprocessor rescans the replacement list to expand additional macros within it. If the preprocessor finds the name of the macro being expanded while rescanning—including the rescanning of nested macro expansions within the replacement list—it won't expand the name again. Furthermore, if macro expansion results in a fragment of program text that's identical to a preprocessing directive, that fragment won't be treated as a preprocessing directive.

During macro expansion, a repeated parameter name in the replacement list will be replaced multiple times by the argument given in the invocation. This can have surprising effects if the argument to the macro invocation involves side effects, as shown in Table 9-9. This problem is explained in detail in CERT C rule PRE31-C (Avoid side effects in arguments to unsafe macros).

Table 9-9: Unsafe Macro Expansion

Original source	Resulting translation unit

`#define bad_abs(x) (x >= 0 ? x : -x)`
 	
`int func(int i) {`
`  return bad_abs(i++);`
`}`
	`int func(int i) {`
	`  return (i++ >= 0 ? i++ : -i++);`
	`}`

In the macro definition in Table 9-9, each instance of the macro parameter `x` is replaced by the macro invocation argument `i++`, causing `i` to be incremented twice in a way that a programmer or reviewer reading the

original source code can easily overlook. Parameters like `x` in the replacement list, as well as the replacement list itself, should usually be fully parenthesized as in `((x) >= 0 ? (x) : -(x))` to prevent portions of the argument `x` from associating with other elements of the replacement list in an unexpected way.

Another potential surprise is that a comma in a function-like macro invocation is always interpreted as a macro argument delimiter. The C Standard `ATOMIC_VAR_INIT` macro, which allows you to initialize an arbitrary atomic variable, demonstrates the danger (Table 9-10). This code fails to translate because the comma in `ATOMIC_VAR_INIT({1, 2})` is treated as a function-like macro argument delimiter, causing the preprocessor to interpret the macro as having two syntactically invalid arguments `{1` and `2}` instead of a single, valid argument `{1, 2}`.[13]

Table 9-10: `ATOMIC_VAR_INIT` Macro

Original sources	Resulting translation unit
stdatomic.h `#define ATOMIC_VAR_INIT(value) (value)`	`<error>`

foo.c
`#include <stdatomic.h>`
 	
`struct S {`
`  int x, y;`
`};`
 	
`_Atomic struct S val = ATOMIC_VAR_INIT({1, 2});`	

Type-Generic Macros

The C programming language doesn't allow you to overload functions based on the types of the parameters passed to the function, as you can in other languages such as Java and C++. However, you might sometimes need to alter the behavior of an algorithm based on the types of the arguments involved. For example, `<math.h>` has three `sin` functions (`sin`, `sinf`, and `sinl`) because each of the three floating-point types (`double`, `float`, and `long double`, respectively) has a different precision. Using generic selection expressions, you can define a single function-like identifier that delegates to the correct underlying implementation based on the argument type when called.

A *generic selection expression* maps the type of its unevaluated operand expression to an associated expression. If none of the associated types match, it can optionally map to a default expression. You can use *type-generic macros* (macros that include generic selection expressions) to make your code more readable. In Table 9-11, we define a type-generic macro to select the correct variant of the `sin` function from `<math.h>`.

13. This usability issue is one of the reasons the `ATOMIC_VAR_INIT` macro was deprecated in C17.

Table 9-11: Generic Selection Expression as a Macro

Original source	Resulting `_Generic` resolution

`#define sin(X) _Generic((X), \`
`  float: sinf, \`
`  double: sin, \`
`  long double: sinl \`
`)(X)`
 	
`int main(void) {`
`  float f = sin(1.5708f);`
`  double d = sin(3.14159);`
`}`
	`int main(void) {`
	`  float f = sinf(1.5708f);`
	`  double d = sin(3.14159);`
	`}`

The controlling expression (`X`) of the generic selection expression is unevaluated; the type of the expression selects a function from the list of `type : expr` mappings. The generic selection expression picks one of these function designators (either `sinf`, `sin`, or `sinl`) and then executes it. In this example, the argument type in the first call to `sin` is `float` so the generic selection resolves to `sinf`, and the argument type in the second call to `sin` is `double` so this resolves to `sin`. Because this generic selection expression has no `default` association, an error occurs if the type of (`X`) doesn't match any of the associated types. If you include a default association for a generic selection expression, it will match every type not already used as an association, including types you might not expect, such as pointers or structure types.

Type-generic macro expansion can be difficult to use when the resulting value type depends on the type of an argument to the macro, as with the `sin` example in Table 9-11. For example, it would be a mistake to call the `sin` macro and assign the result to an object of a specific type, or pass its result as an argument to `printf`, because the necessary object type or format specifier will depend on whether `sin`, `sinf`, or `sinl` is called. Examples of type-generic macros for math functions can be found in the C Standard Library `<tgmath.h>` header.

Predefined Macros

Some macros are defined automatically by the implementation without requiring you to include a header file. These macros are called *predefined macros* because they're implicitly defined by the preprocessor rather than explicitly defined by the programmer. For example, the C Standard defines numerous macros that you can use to interrogate the compilation environment or provide basic functionality. Some other aspects of the implementation (such as the compiler or the compilation target operating system) also automatically define macros. Table 9-12 lists some of the common macros defined by the C Standard. You can obtain a full list of predefined macros from Clang or GCC by passing the `-E -dM` flags to these compilers. You should check your implementation's documentation for more information.

Table 9-12: Predefined Macros

Macro name	Replacement and purpose
`__DATE__`	A string literal of the date of translation of the preprocessing translation unit in the form *`Mmm dd yyyy`*.
`__TIME__`	A string literal of the time of translation for the preprocessing translation unit in the form *`hh:mm:ss`*.
`__FILE__`	A string literal representing the presumed filename of the current source file.
`__LINE__`	An integer constant representing the presumed line number of the current source line.
`__STDC__`	The integer constant 1 if the implementation conforms to the C Standard.
`__STDC_HOSTED__`	The integer constant 1 if the implementation is a hosted implementation, or the integer constant 0 if it is stand-alone. This macro is conditionally defined by the implementation.
`__STDC_VERSION__`	The integer constant representing the version of the C Standard the compiler is targeting, such as `201710L` for the C17 standard.
`__STDC_ISO_10646__`	An integer constant of the form *`yyyymmL`*. This macro is conditionally defined by the implementation. If this symbol is defined, every character in the Unicode required set, when stored in an object of type `wchar_t`, has the same value as the short identifier of that character.
`__STDC_UTF_16__`	The integer constant 1 if values of type `char16_t` are UTF-16 encoded. This macro is conditionally defined by the implementation.
`__STDC_UTF_32__`	The integer constant 1 if values of type `char32_t` are UTF-32 encoded. This macro is conditionally defined by the implementation.
`__STDC_NO_ATOMICS__`	The integer constant 1 if the implementation doesn't support atomic types, including the `_Atomic` type qualifier, and the `<stdatomic.h>` header. This macro is conditionally defined by the implementation.
`__STDC_NO_COMPLEX__`	The integer constant 1 if the implementation doesn't support complex types or the `<complex.h>` header. This macro is conditionally defined by the implementation.
`__STDC_NO_THREADS__`	The integer constant 1 if the implementation doesn't support the `<threads.h>` header. This macro is conditionally defined by the implementation.
`__STDC_NO_VLA__`	The integer constant 1 if the implementation doesn't support variable-length arrays or variably modified types. This macro is conditionally defined by the implementation.

Summary

In this chapter, you learned about some of the features provided by the preprocessor. You learned how to include fragments of program text in a translation unit, conditionally compile code, and generate diagnostics on demand. You then learned how to define and undefine macros, how macros are invoked, and about macros that are predefined by the implementation.

In the next chapter, you'll learn how to structure your program into more than one translation unit to create more maintainable programs.

10

PROGRAM STRUCTURE

with Aaron Ballman

Any real-world system is made up of multiple components, such as source files, header files, and libraries. Many also contain resources including images, sounds, and configuration files. Composing a program from smaller logical components is a good software engineering practice, because these components are easier to manage than a single large file. In this chapter, you'll learn how to structure your program into multiple units consisting of both source and include files. You'll also learn how to link multiple object files together to create libraries and executable files.

Principles of Componentization

Nothing prevents you from writing your entire program within the `main` function of a single source file. However, as the function grows in size, this approach will quickly become unmanageable. For this reason, it makes sense to decompose your program into a collection of components that exchange information across a shared boundary, or *interface*. Organizing source code into components makes it easier to understand, and it allows you to reuse the code elsewhere in the program, or even with other programs.

Understanding how best to decompose a program typically requires experience. Many of the decisions programmers make are driven by performance. For example, you may need to minimize communication over a high-latency interface. Or, a client may handle input field validation from the user interface so that it doesn't require a round trip to the server. In this section, we'll cover some principles of component-based software engineering.

Coupling and Cohesion

In addition to performance, the aim of a well-structured program is to achieve desirable properties like low coupling and high cohesion. *Cohesion* is a measure of the commonality between elements of a programming interface. Assume, for example, that a header file exposes functions for calculating the length of a string, calculating the tangent of a given input value, and creating a thread. This header file has low cohesion, because the exposed functions are unrelated to each other. Conversely, a header file that exposes functions to calculate the length of a string, concatenate two strings together, and search for a substring within a string has high cohesion, because all of the functionality is related. This way, if you need to work with strings, you need only to include the string header file. Similarly, related functions and type definitions that form a public interface should be exposed by the same header file to provide a highly cohesive interface of limited functionality. We'll discuss public interfaces further in "Data Abstractions" on page 187.

Coupling is a measure of the interdependency of programming interfaces. For example, a tightly coupled header file can't be included in a program by itself; instead, it must be included with other header files in a specific order. You may couple interfaces for a variety of reasons, such as a mutual reliance on data structures, interdependence between functions, or the use of a shared global state. But when interfaces are tightly coupled, modifying program behavior becomes difficult, because changes can have a ripple effect across the system. You should always strive to have loose coupling between interface components, regardless of whether they're members of a public interface or implementation details of the program.

By separating your program logic into distinct, highly cohesive components, you make it easier to reason about the components and test the program (because you can verify the correctness of each component independently). The end result is a more maintainable, less buggy system.

Code Reuse

Code reuse is the practice of implementing functionality only once, and then reusing it in various parts of the program without duplicating the code. Code duplication can lead to subtly unexpected behavior, oversized and bloated executables, and increased maintenance costs. And anyway, why write the same code more than once?

Functions are the lowest-level reusable units of functionality. Any logic that you might repeat more than once is a candidate for encapsulating in a function. If the functionality has only minor differences, you can frequently parameterize the function to serve multiple purposes. Each function should perform work that isn't duplicated by any other function. You can then compose individual functions to solve increasingly sophisticated problems.

Packaging reusable logic into functions can improve maintainability and eliminate defects. For example, though you could determine the length of a null-terminated string by writing a simple `for` loop, it's more maintainable to use the `strlen` function from the C Standard Library. Because other programmers are already familiar with the `strlen` function, they'll have an easier time understanding what that function is doing than what the `for` loop is doing. Furthermore, if you reuse existing functionality, you're less likely to introduce behavioral differences compared to ad hoc implementations, and you make it easier to globally replace the functionality with a better-performing algorithm or more secure implementation.

When designing functional interfaces, a balance must be struck between *generality* and *specificity*. An interface that's specific to the current requirement may be lean and effective, but hard to modify when requirements change. A general interface might allow for future requirements, but be cumbersome for foreseeable needs.

Data Abstractions

A *data abstraction* is any reusable software component that enforces a clear separation between the abstraction's public interface and the implementation details. The *public interface* for each data abstraction includes the data-type definitions, function declarations, and constant definitions required by the users of the data abstraction and is placed in header files. The *implementation details* of how the data abstraction is coded, as well as any private utility functions, are hidden within a source file, or a header file that's in a separate location from the public interface header files. This separation of the public interface from the private implementation allows you to change the implementation details without breaking code that depends on your component.

Header files typically contain function declarations and type definitions for the component. For example, the C Standard Library `<string.h>` provides the public interface for string-related functionality, while `<threads.h>` provides utility functions for threading. This logical separation has low coupling and high cohesion. This approach makes it easier to access only the specific components you need, and reduces compile times and the likelihood of name collisions. You don't need to know anything about threading APIs, for example, if all you need is the `strlen` function.

Another consideration is if you should explicitly include the headers required by your header file, or require the users of the header file to include them first. In terms of producing data abstractions, it is a good idea for headers to be self-contained and include the headers they use. Not doing so is more of a burden on the users of the abstraction, and also leaks implementation details about the data abstraction. Examples in this book do not always follow this practice, in an attempt to keep these file listings concise.

Source files implement the functionality declared by a given header file or the application-specific program logic used to perform whatever actions are needed for a given program. For example, if you have a *network.h* header file that describes a public interface for network communications, you may have a *network.c* source file (or *network_win32.c* for Windows only and *network_linux.c* for Linux only) that implements the network communication logic.

It's possible to share implementation details between two source files by using a header file, but the header file should be placed in a distinct location from the public interface so as not to accidentally expose implementation details.

A collection is a good example of a data abstraction that separates the basic functionality from the implementation or underlying data structure. A *collection* represents a group of data elements and supports operations such as adding elements to the collection, removing data elements from the collection, and checking to see if the collection contains a specific data element. There are many ways to implement a collection. For example, a collection of data elements may be represented as a flat array, a binary tree, a directed (possibly acyclic) graph, or a different structure. The choice of data structure can impact an algorithm's performance, depending on what kind of data you're representing and how much data there is to represent. For example, a binary tree may be a better abstraction for a large amount of data that needs good lookup performance, whereas a flat array is likely a better abstraction for a small amount of data of fixed size. Separating the interface of the collection data abstraction from the implementation of the underlying data structure allows the implementation to change without requiring changes to code that relies on the collection interface.

Opaque Types

Data abstractions are most effective when used with opaque data types that hide information. In C, *opaque* (or *private*) data types are those expressed using an incomplete type, such as a forward-declared structure type. An *incomplete type* is a type that describes an identifier but lacks information needed to determine the size of objects of that type or their layout. Hiding internal-only data structures discourages programmers who use the data abstraction from writing code that depends on implementation details, which may change. The incomplete type is exposed to users of the data abstraction, while the fully defined type is accessible only to the implementation.

Let's say that we want to implement a collection that supports a limited number of operations, such as adding an element, removing an element,

and searching for an element. The following example implements `collection_type` as an opaque type, hiding the implementation details of the data type from the library's user. To accomplish this, we create two header files: an external *collection.h* header file included by the user of the data type, and an internal file included only in files that implement the functionality of the data type.

In the external *collection.h* header file, the `collection_type` data type is defined as an instance of `struct collection_type`, which is an incomplete type:

```
typedef struct collection_type collection_type;
// Function declarations
extern errno_t create_collection(collection_type **result);
extern void destroy_collection(collection_type *col);
extern errno_t add_to_collection(collection_type *col, const void *data, size_t byteCount);
extern errno_t remove_from_collection(collection_type *col, const void *data, size_t byteCount);
extern errno_t find_in_collection(const collection_type *col, const void *data,
  size_t byteCount);
//---snip---
```

The `collection_type` identifier is aliased to `struct collection_type` (an incomplete type). Consequently, functions in the public interface must accept a pointer to this type, instead of an actual value type, because of the constraints placed on the use of incomplete types in C.

In the internal header file, `struct collection_type` is fully defined but not visible to a user of the data abstraction:

```
struct node_type {
  void *data;
  size_t size;
  struct node_type *next;
};

struct collection_type {
  size_t num_elements;
  struct node_type *head;
};
```

Modules that implement the abstract data type include both the external and internal definitions, whereas users of the data abstraction include only the external *collection.h* file. This allows the implementation of the `collection_type` data type to remain private.

Executables

In Chapter 9, you learned that the compiler is a pipeline of translation phases, and that the compiler's ultimate output is object code. The last phase of translation, called the *link phase*, takes the object code for all of the translation units in the program and links it together to form a final executable. This can be an executable that a user can run, such as *a.out* or *foo.exe*, a library, or a more specialized program such as a device driver

or a firmware image (machine code to be burned onto a ROM). Linking allows you to separate your code into distinct source files that can be compiled independently, which helps you build reusable components.

Libraries are executable components that cannot be executed independently. Instead, you can incorporate a library into executable programs. You can invoke the functionality of the library by including the library's header files in your source code and calling the declared functions. The C Standard Library is an example of a library—you include the header files from the library, but you do not directly compile the source code that implements the library functionality. Instead, the implementation ships with a prebuilt version of the library code. Libraries allow you to build on the work of others for the generic components of a program so you can focus on developing logic that is unique to your program. For example, when writing a video game, reusing existing libraries should allow you to focus on developing the game logic, not worrying about the details of retrieving user input, network communications, or graphics rendering. Libraries can often allow a program written with one compiler to use code that was built by a different compiler.

Libraries are linked into your application and can be either static or dynamic. A *static library*, also known as an archive, incorporates its machine or object code directly into the resulting executable, which means that a static library is often tied to a specific release of the program. Because a static library is incorporated at link time, the contents of the static library can be further optimized for your program's use of the library. Library code used by the program is available for link-time optimizations, while unused library code can be stripped from the final executable.

A *dynamic library*, also referred to as a shared library or a dynamic shared object, is an executable without the startup routines. It can be packaged with the executable or installed separately but must be available when the executable calls a function provided by the dynamic library. Many modern operating systems will load the dynamic library code into memory once and share it across all the applications that need it. You can replace a dynamic library with different versions as necessary after your application has been deployed. Letting the library evolve separately from the program comes with its own set of benefits and risks. A developer can correct bugs in the library after an application has already shipped without requiring the application to be recompiled, for instance. However, dynamic libraries provide a potential opportunity for a malicious attacker to replace a library with a nefarious one or an end user to accidentally use an incorrect version of the library. It is also possible to make a *breaking change* in a new library release that results in an incompatibility with existing applications that use the library. Static libraries might execute somewhat faster because its object code (binary) is already included in the executable file. Generally speaking, the benefits of using dynamic libraries outweigh the disadvantages.

Each library has one or more header files that contain the public interface to the library, and one or more source files that implement the logic for the library. You can benefit from structuring your code as a collection of libraries even if the components aren't turned into actual libraries. Using

an actual library makes it harder to accidentally design a tightly coupled interface where one component has special knowledge of the internal details of another component.

Linkage

Linkage is a process that controls whether an interface is public or private and determines whether any two identifiers refer to the same entity. C provides three kinds of linkage: external, internal, or none. When a declaration has *external linkage*, identifiers referring to that declaration all refer to the same entity (such as a function or an object) everywhere in the program. When a declaration has *internal linkage*, identifiers referring to that declaration refer to the same entity only within the translation unit containing the declaration. If two translation units both refer to the same internal linkage identifier, they refer to different instances of the entity. If a declaration has *no linkage*, it is a unique entity in each translation unit.

The linkage of a declaration is either explicitly declared or implicitly implied. If you declare an entity at file scope without explicitly specifying `extern` or `static`, the entity is implicitly given external linkage. Identifiers that have no linkage include function parameters, block scope identifiers declared without an `extern` storage class specifier, or enumeration constants.

Listing 10-1 shows examples of declarations of each kind of linkage.

```
static int i; // i is declared with explicit internal linkage.
extern void foo(int j) {
  // foo is declared with explicit external linkage.
  // j has no linkage because it is a parameter.
}
```

Listing 10-1: Examples of external, internal, and no linkage

If you explicitly declare an identifier with the `static` storage class specifier at file scope, it has internal linkage. The `static` keyword gives internal linkage only to file scope entities. Declaring a variable at block scope as `static` creates an identifier with no linkage, but it does give the variable static storage duration. As a reminder, static storage duration means its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup. The different meanings of static when used in different contexts is obviously confusing and consequently a common interview question.

You can create an identifier with external linkage by declaring it with the `extern` storage class specifier. This works only if you haven't previously declared the linkage for that identifier. The `extern` storage class specifier has no effect if a prior declaration gave the identifier linkage.

Declarations with conflicting linkage can lead to undefined behavior; see CERT C rule DCL36-C (Do not declare an identifier with conflicting linkage classifications) for more information.

Table 10-1 shows examples of declarations with explicit and implicit linkage.

Table 10-1: Examples of Implicit and Explicit Linkage

Implicit and explicit linkage

foo.c
`void func(int i) { // Implicit external linkage`
`  // i has no linkage`
`}`
`static void bar(void); // Internal linkage, different bar than bar.c`
`extern void bar(void) {`
`  // bar still has internal linkage because the initial declaration was`
`  // declared as static; the extern specifier has no effect in this case.`
`}`

bar.c
`extern void func(int i); // Explicit external linkage`
 	
`static void bar(void) {  // Internal linkage; different bar than foo.c`
`  func(12); // Calls func from foo.c`
`}`
`int i; // External linkage; does not conflict with i from foo.c or bar.c`
`void baz(int k) { // Implicit external linkage`
`  bar(); // Calls bar from bar.c, not foo.c`
`}`

The identifiers in your public interface should have external linkage so that they can be called from outside their translation unit, while the identifiers that are implementation details should be declared with internal or no linkage. A common approach to achieving this is to declare your public interface functions in a header file with or without using the `extern` storage class specifier (the declarations implicitly have external linkage, but there is no harm in explicitly declaring them with `extern`), and define the public interface functions in a source file in a similar manner.

However, within the source file, all declarations that are implementation details should be explicitly declared `static` to keep them private, accessible to just that source file. You can include the public interface declared within the header file by using the `#include` preprocessor directive to access its interface from another file. A good rule of thumb is that file-scope entities that don't need to be visible outside the file should be declared as `static`. This practice limits the pollution of the global namespace and decreases the chances of surprising interactions between translation units.

Structuring a Simple Program

To learn how to structure a complex, real-world program, let's develop a simple program to determine whether a number is prime. A *prime number* (or a *prime*) is a natural number greater than one that cannot be formed by multiplying two smaller natural numbers. We'll write two separate components: a static library containing the testing functionality and a command line application that provides a user interface for the library.

The *primetest* program accepts a whitespace-delimited list of integer values as input and outputs whether each value is a prime number. If any of the inputs are invalid, the program will output a helpful message explaining how to use the interface.

Before exploring how to structure the program, let's examine the user interface. First, we print the help text for the command line program, as shown in Listing 10-2.

```
// Print command line help text.
static void print_help(void) {
  printf("%s", "primetest num1 [num2 num3 ... numN]\n\n");
  printf("%s", "Tests positive integers for primality. Supports testing ");
  printf("%s [2-%llu].\n", "numbers in the range", ULLONG_MAX);
}
```

Listing 10-2: Printing help text

The `print_help` function consists of three separate calls to the `printf` function that print help text about how to use the command to the standard output.

Next, because the command line arguments are passed to the program as textual input, we define a utility function to convert them to integer values, as shown in Listing 10-3.

```
// Converts a string argument arg to an unsigned long long value referenced by val.
// Returns true if the argument conversion succeeds, and false if it fails.
static bool convert_arg(const char *arg, unsigned long long *val) {
  char *end;

  // strtoll returns an in-band error indicator; clear errno before the call.
  errno = 0;
  *val = strtoull(arg, &end, 10);

  // Check for failures where the call returns a sentinel value and sets errno.
  if ((*val == ULLONG_MAX) && errno) return false;
  if (*val == 0 && errno) return false;
  if (end == arg) return false;

  // If we got here, we were able to convert the argument. However, we want to allow only
  // values greater than one, so we reject values <= 1.
  if (*val <= 1) return false;
  return true;
}
```

Listing 10-3: Converting a single command line argument

The `convert_arg` function accepts a string argument as input and uses an output parameter to report the converted argument. (An *output parameter* returns a function result to the caller via a parameter instead of the return value, allowing multiple values to be returned.) The function returns `true` if the argument conversion succeeds, and `false` if it fails. The `convert_arg` function uses the `strtoull` function to convert the string to an `unsigned long long`

integer value and takes care to properly handle conversion errors. Additionally, because the definition of a prime number excludes 0, 1, and negative values, the `convert_arg` function treats those as invalid inputs.

We use the `convert_arg` utility function in the `convert_command_line_args` function, shown in Listing 10-4, which loops over all of the command line arguments provided and attempts to convert each argument from a string to an integer.

```
static unsigned long long *convert_command_line_args(int argc,
                                                     const char *argv[],
                                                     size_t *num_args) {
  *num_args = 0;

  if (argc <= 1) {
    // No command line arguments given (the first argument is the name of the
    // program being executed).
    print_help();
    return NULL;
  }

  // We know the maximum number of arguments the user could have passed,
  // so allocate an array large enough to hold all of the elements. Subtract
  // one for the program name itself. If the allocation fails, treat it as
  // a failed conversion (it is okay to call free(NULL)).
  unsigned long long *args =
      (unsigned long long *)malloc(sizeof(unsigned long long) * (argc - 1));
  bool failed_conversion = (args == NULL);
  for (int i = 1; i < argc && !failed_conversion; ++i) {
    // Attempt to convert the argument into an integer. If we
    // couldn't convert it, set failed_conversion to true.
    unsigned long long one_arg;
    failed_conversion |= !convert_arg(argv[i], &one_arg);
    args[i - 1] = one_arg;
  }

  if (failed_conversion) {
    // Free the array, print the help, and bail out.
    free(args);
    print_help();
    return NULL;
  }

  *num_args = argc - 1;
  return args;
}
```

Listing 10-4 Processing all of the command line arguments

If any argument fails to convert, it calls the `print_help` function to report the proper command line usage to the user, and then returns a null pointer. This function is responsible for allocating a sufficiently large buffer to hold the array of integers. It also handles all error conditions, such as running

out of memory or failing to convert an argument. If the function succeeds, it returns an array of integers to the caller and writes the converted number of arguments into the `num_args` parameter. The returned array is allocated storage and must be deallocated when no longer needed.

There are several ways to determine whether a number is prime. The naive approach is to test a value *N* by determining whether it is evenly divisible by [2..*N*-1]. This approach has poor performance characteristics as the value of *N* gets larger. Instead, we'll use one of the many algorithms designed for testing primality. Listing 10-5 shows a nondeterministic implementation of the Miller-Rabin primality test that's suitable for quickly testing whether a value is probably prime (Schoof 2008). Please see the Schoof paper for an explanation of the mathematics behind the Miller-Rabin primality test algorithm.

```
static unsigned long long power(unsigned long long x, unsigned long long y,
                                unsigned long long p) {
  unsigned long long result = 1;
  x %= p;

  while (y) {
    if (y & 1) result = (result * x) % p;
    y >>= 1;
    x = (x * x) % p;
  }
  return result;
}

static bool miller_rabin_test(unsigned long long d, unsigned long long n) {
  unsigned long long a = 2 + rand() % (n - 4);
  unsigned long long x = power(a, d, n);

  if (x == 1 || x == n - 1) return true;

  while (d != n - 1) {
    x = (x * x) % n;
    d *= 2;

    if (x == 1) return false;
    if (x == n - 1) return true;
  }
  return false;
}
```

Listing 10-5: Miller-Rabin primality test algorithm

The interface to the Miller-Rabin primality test is the `is_prime` function shown in Listing 10-6. This function accepts two arguments: the number to test (`n`) and the number of times to perform the test (`k`). Larger values of `k` provide a more accurate result at the expense of worse performance. We'll place the algorithm from Listing 10-5 in a static library, along with the `is_prime` function, which will provide the library's public interface.

```
bool is_prime(unsigned long long n, unsigned int k) {
  if (n <= 1 || n == 4) return false;
  if (n <= 3) return true;

  unsigned long long d = n - 1;
  while (d % 2 == 0) d /= 2;

  for (; k != 0; --k) {
    if (!miller_rabin_test(d, n)) return false;
  }
  return true;
}
```

Listing 10-6: Interface to the Miller-Rabin primality test algorithm

Finally, we need to compose these utility functions into a program. Listing 10-7 shows the implementation of the `main` function. It uses a fixed number of iterations of the Miller-Rabin test and reports whether the input values are probably prime or definitely not prime. It also handles deallocating the memory allocated by `convert_command_line_args`.

```
int main(int argc, char *argv[]) {
  size_t num_args;
  unsigned long long *vals = convert_command_line_args(argc, argv, &num_args);

  if (!vals) return EXIT_FAILURE;

  for (size_t i = 0; i < num_args; ++i) {
    printf("%llu is %s.\n", vals[i],
           is_prime(vals[i], 100) ? "probably prime" : "not prime");
  }

  free(vals);
  return EXIT_SUCCESS;
}
```

Listing 10-7: The `main` function

The `main` function calls the `convert_command_line_args` function to convert the command line arguments into an array of `unsigned long long` integers. For each argument in this array, the program loops, calling `is_prime` to determine whether each value is probably prime or not prime using the Miller-Rabin primality test implemented by the `is_prime` function.

Now that we've implemented the program logic, we'll produce the required build artifacts. Our goal is to produce a static library containing the Miller-Rabin implementation and a command line application driver.

Building the Code

Create a new file named *isprime.c* with the code from Listing 10-5 and 10-6 (in that order), adding the `#include` directives for `"isprime.h"` and `<stdlib.h>`

at the top of the file. The quotes and angle brackets surrounding the header files are important for telling the preprocessor where to search for those files, as discussed in Chapter 9. Next, create a header file named *isprime.h* with the code from Listing 10-8 to provide the public interface for the static library, with a header guard.

```
#ifndef PRIMETEST_IS_PRIME_H
#define PRIMETEST_IS_PRIME_H

#include <stdbool.h>

bool is_prime(unsigned long long n, unsigned k);

#endif // PRIMETEST_IS_PRIME_H
```

Listing 10-8: The public interface for the static library

Create a new file named *driver.c* with the code from Listings 10-2, 10-3, 10-4, and 10-7 in that order, adding the `#include` directives for the following: `"isprime.h"`, `<assert.h>`, `<errno.h>`, `<limits.h>`, `<stdbool.h>`, `<stdio.h>`, and `<stdlib.h>` at the top of the file. All three files are in the same directory in our example, but in a real-world project, you would likely put the files in different directories, depending on the conventions of your particular build system. Create a local directory named *bin*, which is where the build artifacts from this example will be created.

We use Clang to create the static library and executable program, but both GCC and Clang support the command line arguments in the example, so either compiler will work. First, compile both of the C source files into object files placed in the *bin* directory:

```
% clang -c -std=c17 -Wall -Wextra -pedantic -Werror isprime.c -o bin/isprime.o
% clang -c -std=c17 -Wall -Wextra -pedantic -Werror driver.c -o bin/driver.o
```

If you execute the command and get an error such as

```
unable to open output file 'bin/isprime.o': 'No such file or directory'
```

then create the local *bin* directory and try the command again. The `-c` flag instructs the compiler to compile the source into an object file without invoking the linker to produce executable output. We'll need the object files to create a library. The `-o` flag specifies the pathname of the output file.

After executing the commands, the *bin* directory should contain two object files: *isprime.o* and *driver.o*. These files contain the object code for each translation unit. You could link them together directly to create the executable program. However, in this case, we'll make a static library (also called an *archive* for historical reasons). To do this, execute the `ar` command to generate the static library named *libPrimalityUtilities.a* in the *bin* directory:

```
% ar rcs bin/libPrimalityUtilities.a bin/isprime.o
```

The `r` option instructs the `ar` command to replace any existing files in the archive with the new files, the `c` option creates the archive, and the `s` option writes an object-file index into the archive (which is equivalent to running the `ranlib` command). This creates a single archive file that's structured to allow retrieval of the original object files used to create the archive, similar to a compressed tarball or ZIP file. By convention, static libraries on Linux systems are prefixed with *lib* and have a *.a* file extension.

You can now link the driver object file to the *libPrimalityUtilities.a* static library to produce an executable named *primetest*. This can be accomplished either by invoking the compiler without the `-c` flag, which invokes the default system linker with the appropriate arguments, or by invoking the linker directly. Invoke the compiler to use the default system linker as follows:

```
% clang bin/driver.o -Lbin -lPrimalityUtilities -o bin/primetest
```

The `-L` flag instructs the linker to look in the local *bin* directory for libraries to link, and the `-l` flag instructs the linker to link the *libPrimalityUtilities.a* library to the output. Omit the *lib* prefix and the *.a* suffix from the command line argument, because the linker adds them implicitly. For example, to link against the *libm* math library, you specify `-lm` as the link target. As with compiling source files, the output of the linked files is specified by the `-o` flag.

You can now test the program to see whether values are probably prime or definitely not prime. Be sure to try out cases like negative numbers, known prime and nonprime numbers, and incorrect input, as shown in Listing 10-9.

```
% ./bin/primetest 899180
899180 is not prime
% ./bin/primetest 8675309
8675309 is probably prime
% ./bin/primetest 0
primetest num1 [num2 num3 ... numN]

Tests positive integers for primality. Supports testing numbers in the range
[2-18446744073709551615].
```

Listing 10-9: Running the primetest *program with sample input*

The number 8,675,309 is prime.

Summary

In this chapter, you learned about the benefits of loose coupling and high cohesion, data abstractions, and code reuse. Additionally, you learned about related language constructs such as opaque data types and linkage. You also learned some best practices on how to structure the code in your projects, and saw an example of building a simple program with different types of executable components. In the next chapter, we'll examine tools and techniques for creating high-quality systems, including assertions, debugging, testing, and static and dynamic analysis.

11

DEBUGGING, TESTING, AND ANALYSIS

This chapter describes tools and techniques for producing correct, effective, safe, secure, and robust programs, including static (compile-time) and runtime assertions, debugging, testing, static analysis, and dynamic analysis. The chapter also discusses which compiler flags are recommended for use in different phases of the software development process.

Assertions

You use an *assertion* to verify that a specific assumption you made during the implementation of your program remains valid. An assertion is a function with a Boolean value, known as a *predicate*, which expresses a logical proposition about a program. C supports static assertions that can be

checked at compile time using `static_assert`, and runtime assertions that are checked during program execution using `assert`. Both the `assert` and `static_assert` macros are defined in the `<assert.h>` header.

Static Assertions

Static assertions can be expressed using the `static_assert` macro as follows:

```
static_assert(integer-constant-expression, string-literal);
```

If the value of the integer constant expression is not equal to 0, the `static_assert` declaration has no effect. If the integer constant expression is equal to 0, the compiler will produce a diagnostic message with the text of the string literal you specify.

You can use static assertions to validate assumptions at compile time, such as specific implementation-defined behaviors. Any change in implementation-defined behavior will then be diagnosed at compile time.

Let's look at three examples of using static assertions. First, in Listing 11-1, we use `static_assert` to verify that `struct packed` lacks padding bytes.

```
#include <assert.h>

struct packed {
  unsigned int i;
  char *p;
};

static_assert(
  sizeof(struct packed) == sizeof(unsigned int) + sizeof(char *),
  "struct packed must not have any padding"
);
```

Listing 11-1: Using `static_assert` to verify that a structure lacks padding bytes

The predicate for the static assertion in this example verifies that the size of the packed structure is the same as the combined size of its `unsigned int` and `char *` members. Because a static assertion is a declaration, it can appear at file scope, immediately following the definition of the `struct` whose property it asserts.

Next, the `clear_stdin` function, shown in Listing 11-2, calls the `getchar` function to read characters from `stdin` until the end of the file is reached. Each character is obtained as an `unsigned char` converted to an `int`. It's common practice to compare the character returned by the `getchar` function with `EOF`, often in a `do...while` loop, to determine when all the available characters have been read. For this function loop to work correctly, the terminating condition must be able to differentiate between a character and `EOF`. However, the C Standard allows for `unsigned char` and `int` to have the same range, meaning that on some implementations, this test for `EOF` could return false positives, in which case the `do...while` loop may terminate early.

Because this is an unusual condition, you can use `static_assert` to validate that the `do...while` loop can properly distinguish between valid characters and `EOF`.

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

void clear_stdin(void) {
  int c;

  do {
    c = getchar();
    static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation");
  } while (c != EOF);
}
```

Listing 11-2: Using `static_assert` to verify integer sizes

In this example, the static assertion verifies that the maximum `unsigned char` value `UCHAR_MAX` is less than the maximum `unsigned int` value `UINT_MAX`. The static assertion is placed near the code that depends on this assumption being true, so that you can easily locate the code that will need to be repaired if the assumption is violated. Because static assertions are evaluated at compile time, placing them within executable code has no impact on the runtime efficiency of the program. See the CERT C rule FIO34-C (Distinguish between characters read from a file and `EOF` or `WEOF`) for more information on this topic.

Finally, in Listing 11-3, we use `static_assert` to perform bounds checking at compile time. This code snippet uses `strcpy` to copy a constant string `prefix` to a statically allocated array `str`. The static assertion ensures that `str` will have sufficient space to store at least one additional character for an error code following the call to `strcpy`.

```
static const char prefix[] = "Error No: ";
#define ARRAYSIZE 14
char str[ARRAYSIZE];

// Ensure that str has sufficient space to store at least one additional
// character for an error code
static_assert(
  sizeof(str) > sizeof(prefix),
  "str must be larger than prefix"
);
strcpy(str, prefix);
```

Listing 11-3: Using `static_assert` to perform bounds checking

This assumption may become invalid if a developer, for example, were to reduce `ARRAYSIZE`, or change the prefix string to `"Error Number: "` during maintenance. Having added the static assertion, the maintainer would now

be warned about the problem. Remember that the string literal is a message for the developer or maintainer, and not an end user of the system. It's intended to provide information useful to debugging.

Runtime Assertions

The assert macro injects runtime diagnostic tests into programs. It's defined in the <assert.h> header file and takes a scalar expression as a single argument:

```
#define assert(scalar-expression) /* implementation defined */
```

The assert macro is implementation defined. If the scalar expression is equal to 0, the macro expansion typically writes information about the failing call (including the argument text, the name of the source file __FILE__, the source line number __LINE__, and the name of the enclosing function __func__) to the standard error stream stderr. After writing this information to stderr, the assert macro calls the abort function.

The dup_string function shown in Listing 11-4 uses runtime assertions to check that the size argument is less than or equal to LIMIT and that str is not a null pointer.

```
void *dup_string(size_t size, char *str ) {
  assert(size <= LIMIT);
  assert(str != NULL);
  //---snip---
}
```

Listing 11-4: Using assert to verify program conditions

The messages from these assertions might take the following form:

```
Assertion failed: size <= LIMIT, function dup_string, file foo.c, line 122.
Assertion failed: str != NULL, function dup_string, file foo.c, line 123.
```

The implicit assumption is that the caller validates arguments before calling dup_string so that the function is never called with invalid arguments. The runtime assertions are then used to validate this assumption during the development and test phases.

The assertion's predicate expression is often reported in a failed assertion message, which allows you to use && on a string literal with the assertion predicate to generate additional debugging information when an assertion fails. Doing so is always safe because string literals in C can never have a null pointer value. For example, we can rewrite the assertions in Listing 11-4 to have the same functionality but provide additional context when the assertion fails, as shown in Listing 11-5.

```
void *dup_string(size_t size, char *str ) {
  assert(size <= LIMIT && "size is larger than the expected limit");
```

```
  assert(str != NULL && "the caller must ensure str is not null");
  //---snip---
}
```

Listing 11-5: Using `assert` with additional contextual information

You should disable assertions before code is deployed by defining the `NDEBUG` macro (typically as a flag passed to the compiler). If `NDEBUG` is defined as a macro name at the point in the source file where `<assert.h>` is included, the `assert` macro is defined as follows:

```
#define assert(ignore) ((void)0)
```

The reason the macro does not expand empty is because if it did, then code such as

```
assert(thing1) // missing semi-colon
assert(thing2);
```

would compile in release mode but not in debug mode. The reason it expands to `((void) 0)` rather than just `0` is to prevent warnings about statements with no effect. The `assert` macro is redefined according to the current state of `NDEBUG` each time that `<assert.h>` is included.

Use static assertions to check assumptions that can be checked at compile time and runtime assertions to detect invalid assumptions during testing. Because runtime assertions are typically disabled before deployment, avoid using them to check for conditions that can come up during normal operations, such as the following:

- Invalid input
- Errors opening, reading, or writing streams
- Out-of-memory conditions from dynamic allocation functions
- System call errors
- Invalid permissions

You should instead implement these checks as normal error-checking code that's always included in the executable. Assertions should only be used to validate preconditions, post-conditions, and invariants designed into the code (programming errors).

Compiler Settings and Flags

Compilers often don't enable optimization or security hardening by default. Instead, you can enable optimization, error detection, and security hardening using build flags (Weimer 2018). I'll recommend specific flags for GCC, Clang, and Visual C++ in the next section, after first describing how and why you might want to use them.

You should select your build flags based on what you're trying to accomplish. Distinct phases of software development call for different sets of build flags:

Analysis During analysis, you're trying to get your code to compile. Dealing with numerous diagnostics at this stage can seem bothersome, but is much better than having to find these problems through debugging and testing, or not finding them until after the code has shipped. During the analysis phase, you should use compiler options that maximize diagnostics to help you eliminate as many defects as possible.

Debugging During debugging, you're typically trying to figure out why your code isn't working, so you should use a set of compiler flags that includes debug information, allows assertions to be useful, injects runtime instrumentation to detect errors, and enables a quick turnaround time for the inevitable edit-compile-debug cycle.

Testing During testing, you may want to disable debug information except for symbol names so that you can get good stack traces for crashes and leave assertions enabled. You may also want to start testing optimized builds. These settings may also be used in the beta version of a product to help isolate any defects discovered during beta testing.

Acceptance testing/deployment The final phase is to build the code for deployment to its operational environment. Before deploying the system, make sure you adequately test your build configuration, because using a different set of compilation flags can trigger new defects—for example, as the result of timing effects from running optimized code.

I'll now recommend some specific flags you might want to use for your compiler and software development phase.

GCC and Clang

Table 11-1 lists recommended compiler and linker flags for both GCC and Clang, as well as differences between the two compilers. You can find documentation for compiler flags in the GCC manual (*https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html*) and the Clang Compiler User's Manual (*https://clang.llvm.org/docs/UsersManual.html#command-line-options*).

Table 11-1: Recommended Compiler and Linker Flags for GCC and Clang

Flag	Purpose
`-D_FORTIFY_SOURCE=2`	Detect runtime buffer overflows
`-fpie -Wl,-pie`	Needed to enable full ASLR for executables
`-fpic -shared`	Disable text relocations for shared libraries
`-g3`	Generate abundant debugging information
`-O2`	Optimize your code for speed/space efficiency
`-Wall`	Turn on recommended compiler warnings

Flag	Purpose
`-Werror`	Turn warnings into errors
`-std=c17`	Specify the language standard
`-pedantic`	Issue warnings demanded by strict conformance to the standard

-O Optimization

The uppercase letter `-O` flag controls *compiler optimization*. Most optimizations are disabled at `-O0` or if an `-O` level is not set on the command line. If optimizations are disabled, the compiler attempts to speed compilation and simplify debugging. The `-Og` level optimizes the code for debugging, suppressing many optimization passes, and is a better choice than `-O0` for producing debuggable code.

When you're ready to deploy your application or submit it for acceptance testing, you'll want to optimize your code. Turning on optimization instructs the compiler to improve the performance or code size at the expense of compilation time and your ability to debug the program. The `-O2` level performs almost all optimizations that don't involve a space-speed trade-off and is generally recommended for production-level code. The `-O2` level or higher is also required by `FORTIFY_SOURCE`.

When initially compiling your code (for example, during the analysis phase), you may want to use the `-O2` flag to enable diagnostics that are only performed for optimized builds when building with GCC. Do not enable sanitization (discussed in "AddressSanitizer" on page 19) during analysis because the inserted runtime instrumentation can cause false positives.

The `-O1` level performs fast optimizations that don't significantly increase build times, which might be useful for testing.

-glevel Debugging

The `-g`*`level`* flag produces debugging information in the operating system's native format. You can specify how much information to produce by setting the debug *`level`*. The default level is `-g2`. Level 3 (`-g3`) includes extra information, such as all the macro definitions present in the program. Level 3 also allows you to expand macros in debuggers that support the capability.

-Wall Warnings

The `-Wall` flag enables a useful set of warning flags, but not all of them. You can also specify `-Wextra` to enable additional warnings not enabled by `-Wall`.

-Werror Errors

The `-Werror` flag turns all warnings into errors, requiring you to address them before you can begin debugging. This flag simply encourages good programming discipline.

-std=flag Language Standards

The `-std` flag can be used to specify the language standard as `c89`, `c90`, `c99`, `c11`, `c17`, or `c2x`. For GCC, the default, if no C language dialect options are given, is `-std=gnu17`, which provides some extensions to the C language that, on rare occasions, conflict with the C standard. For Clang, the default is `-std=gnu11`. For portability, specify the standard you're using. For access to new language features, specify a recent standard. A good choice (in 2020) is `-std=c17`.

-pedantic Warnings

The `-pedantic` flag issues warnings when code deviates from strict conformance to the standard. This flag is typically used in conjunction with the `-std` flag to improve the portability of the code.

-D_FORTIFY_SOURCE=2 Buffer Overflow Detection

The `_FORTIFY_SOURCE` macro provides lightweight support for detecting buffer overflows in functions that perform operations on memory and strings. Not all types of buffer overflows can be detected with this macro, but compiling your source with `-D_FORTIFY_SOURCE=2` provides an extra level of validation for functions that copy memory and are a potential source of buffer overflows such as `memcpy`, `memset`, `strcpy`, `strcat`, and `sprintf`. Some of the checks can be performed at compile time and result in diagnostics; others occur at runtime and can result in a runtime error.

-fpie -Wl, -pie and -fpic -shared Position Independence

Address space layout randomization (ASLR) is a security mechanism that randomizes the process's memory space to prevent attackers from locating the code they're trying to execute. You can learn more about ASLR and other security mitigations in *Secure Coding in C and C++* (Seacord 2013).

You must specify the `-fpie -Wl, -pie` flags to create position-independent executable programs and to make it possible to enable ASLR for your main program (executable). However, while code emitted for your main program with these options is position-independent, it does use some relocations that cannot be used in shared libraries (dynamic shared objects). For those, use `-fpic`, and link with `-shared` to avoid text relocations on architectures that support position-dependent shared libraries. Dynamic shared objects are always position-independent and therefore support ASLR.

Visual C++

Visual C++ provides a wide assortment of compiler options, many of which are similar to the options available for GCC and Clang.[14] One obvious difference is that Visual C++ generally uses the forward slash (`/`) character

14. For more information on compiler options, see *https://docs.microsoft.com/en-us/cpp/build/reference/compiler-options-listed-by-category/*

instead of a hyphen (-) to indicate a flag. Table 11-2 lists recommended compiler and linker flags for Visual C++.

Table 11-2: Recommended Compiler Flags for Visual C++

Flag	Purpose
`/guard:cf`	Add control flow guard security checks
`/analyze`	Enable static analysis
`/sdl`	Enable security features
`/permissive-`	Specify standards conformance mode to the compiler
`/O2`	Set optimization to level 2
`/W4`	Set compiler warnings to level 4
`/WX`	Turn warnings into errors

Several of these options are similar to options provided by the GCC and Clang compilers. `/O2` is a good optimization level for deployed code, while `/Od` disables optimization, to speed compilation and simplify debugging. `/W4` is a good warning level, especially for new code, and it's roughly equivalent to `-Wall` in GCC and Clang. `/Wall` in Visual C++ is not recommended because it produces a high number of false positives. `/WX` turns warnings into errors and is equivalent to the `-Werror` flag in GCC and Clang. I cover the remaining flags in further detail in the following sections.

/guard:cf Security Checks

When you specify the *control flow guard (CFG)* option, the compiler and linker insert extra runtime security checks to detect attempts to compromise your code. The `/guard:cf` option must be passed to both the compiler and the linker.

/analyze Static Analysis

The `/analyze` flag enables static analysis, which provides information about possible defects in your code. I discuss static analysis in more detail in "Static Analysis" on page 214.

/sdl Security Features

The `/sdl` flag enables additional security features, including treating extra security-relevant warnings as errors, and additional secure code-generation features. It also enables other security features from the Microsoft *Security Development Lifecycle (SDL)*. The `/sdl` flag should be used in all production builds where security is a concern.

/permissive- Standard Conformance

You can use `/permissive-` to help identify and fix conformance issues in your code, thereby improving the correctness and portability of your

code. This option disables permissive behaviors and sets the /Zc compiler options for strict conformance. In the IDE, this option also underlines nonconforming code.

Debugging

I've been programming professionally for 37 years. Once or maybe twice during that time, I've written a program that compiled and ran correctly on the first try. For all the other times, there is debugging.

Let's debug a faulty program. The program shown in Listing 11-6 tests the print_error function but doesn't produce the expected result.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <malloc.h>

errno_t print_error(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum);
  char* msg = malloc(size);
❶ if ((msg != NULL) && (strerror_s(msg, size, errnum) != 0)) {
    fputs(msg, stderr);
    return 0;
  }
  else {
  ❷ fputs("unknown error", stderr);
    return ENOMEM;
  }
}

int main(void) {
  print_error(ENOMEM);
  exit(1);
}
```

Listing 11-6: Printing an error

When this program is run from Visual C++, it outputs the following:

```
unknown error
```

This is not the expected outcome of this test. The ENOMEM macro should generate a string similar to "out of memory". Having the string "unknown error" output probably means that the fputs call in the else clause ❷ executed, which would happen if either the call to malloc or strerror_s failed. There is also a slight possibility that "unknown error" is the actual string returned by the call to strerror_s. We can test this potentiality by changing the string output at ❷ to something else, like "bananarama", and

rerunning the test. When we recompile and rerun the test, we'll see the following output:

```
bananarama
```

We can take this as conclusive evidence that the `else` statement is executing. This suggests that the result of the test at ❶ is false, causing the `else` statement to be executed. We need to test whether this is the case.

Novice programmers have a strong tendency to debug everything by inserting print statements throughout the code, but using a debugger is often much more productive. In Visual C++, you can start the debugger by selecting Start Debugging from the Debug menu. Before doing so, you should set a *breakpoint*, which will mark the point in the execution of the program where you want the debugger to temporarily suspend execution and allow you to examine the state of the program. You set a breakpoint in Visual C++ by clicking to the left of the line number you wish to break at—for example, before the `if` statement at ❶.

Set a breakpoint here and then start debugging. The debugger should stop just before executing the `if` statement. It's a good idea to examine the program state before continuing by looking at the values of some of the local and automatic variables. Specifically, examine the value of `msg` to make sure the call to `malloc` succeeded. This value should appear in either the Autos tab that displays automatic variables or the Locals tab that displays local variables. In either case, you should see something like Table 11-3.

Table 11-3: Values of Automatic Variables, Displayed in the Autos Tab

Name	Value	Type
errnum	12	int
msg	0x00a56120 "ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍýýýý\bP¥"	char *
size	16	unsigned int

In this tab, we can see the values and types of the three automatic variables `errnum`, `msg`, and `size`. Notice that `msg` has a valid address pointing to some uninitialized memory. At this point, these variables all have reasonable values.

The next thing we can do is *single step*, or execute the entire current line of code. Visual C++ has three variations of single step available: Step Into, Step Over, and Step Out.

Step Into progresses the execution of the program to the first line of any function that's being called. This option will continue to dive down into each function call, so long as there is source code available. *Step Over* continues to the next line in the function that's currently executing. *Step Out* continues execution until the currently executing function exits.

In this case, we will select Step Over to see which statement executes next. Control continues to the `fputs` call in the `else` statement ❶, but it's still not clear why. One possible hypothesis is that the call to `strerror_s` is failing somehow, so next we'll try to catch the error.

It may be possible to check the return value of the function in Visual C++, but instead we will temporarily rewrite the `print_error` function to store the return value from `strerror_s` in the automatic variable `status`, as shown in Listing 11-7. We can now easily examine the value stored in this variable.

```
errno_t print_error(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum);
  char* msg = malloc(size);
  if (msg != NULL) {
    errno_t status = strerror_s(msg, size, errnum);
 ❶ if (status != 0) {
      fputs(msg, stderr);
      return ENOMEM;
    }
  }
  else {
    fputs("unknown error", stderr);
    return ENOMEM;
  }
}
```

Listing 11-7: Rewritten `print_error` function

Set the breakpoint at the test for `status != 0` ❶ and examine the value of the `status` automatic variable in the debugger to determine which error has occurred. You should see that `status` has the value 0, indicating that no error occurred, so we can rule out `strerror_s` failing as the problem.

Now that we see that `status` has the value 0, it should become obvious to you that the test at ❶ is inverted, and that we want to print the error messages when the call to `strerror_s` succeeds, not when it fails. You can repair this defect by testing to see if `status` is equal to 0, as follows:

```
    if (status == 0) {
```

Now that we have most likely fixed a bug and restored our overconfidence, we'll rerun this program:

```
Not enough spac
```

We now appear to be taking the correct branch and getting the correct error message, but the error message appears to be truncated. This is still progress (sort of). At times like this, I generally get desperate enough to read the documentation. Section K.3.7.4.2 of the C Standard ("The `strerror_s` function") states the following:

> If the length of the desired string is less than `maxsize`, then the string is copied to the array pointed to by `s`. Otherwise, if `maxsize` is greater than zero, then `maxsize-1` characters are copied from the string to the array pointed to by s, and then `s[maxsize-1]` is

set to the null character. Then, if `maxsize` is greater than 3, then `s[maxsize-2]`, `s[maxsize-3]`, and `s[maxsize-4]` are set to the character period (`.`).

We can now see that things are so much worse than originally believed (they usually are). Because bounds-checked interfaces normally fail if they don't completely copy a string, the `print_error` function assumes that the `strerror_s` function will fail if it can't completely copy the string into the available storage, but apparently this is not the case. However, we are also not seeing the documented behavior of setting the end of the string to `"..."` as required by the standard. This would have resulted in the following output:

```
Not enough s...
```

Checking the specification for the `strerrorlen_s` function, we can see that it returns the number of characters (not including the null character) in the full message string. This explanation makes sense and is consistent with the behavior of the `strlen` function. So perhaps we can solve the problem by adding one additional byte to `size`, as follows:

```
rsize_t size = strerrorlen_s(errnum) + 1;
```

Once again, let's be overly confident and run the program:

```
Not enough space
```

Apparently, our confidence was justified, because the full error message has now been successfully output. However, a defect remains in this version of the `print_error` function. I'll come back to it later, but see if you can find it.

Once the `print_error` function has been successfully debugged, you can restore it to a corrected version of its more compact representation or leave it as it is. If you modify the code again, make sure you retest the program to make sure it's still working. You should also take the time to report any defects in the implementation to the developer.

Unit Testing

Now that we have a "working" implementation of the `print_error` function, it's time to do some unit testing, to confirm our assumption that the function is working. *Unit tests* are small programs that exercise your code. *Unit testing* is a process that validates that each unit of the software performs as designed. A *unit* is the smallest testable part of any software; in C, this is typically an individual function or data abstraction.

You can write simple tests that resemble normal application code (see Listing 11-7, for example), but it's usually beneficial to use a *unit-testing framework*. Several unit-testing frameworks are available, including Google

Test, CUnit, Unity, DejaGnu, and CppUnit. We'll examine the most popular of these (based on the most recent survey of the C development ecosystem by JetBrains): Google Test.

Google Test works for Linux, Windows, or macOS. Tests are written in C++, so you get to learn another (related) programming language for testing purposes. In Google Test, you write assertions to verify the tested code's behavior. Google Test assertions, which are function-like macros, are the real language of the tests. If a test crashes or has a failed assertion, it fails; otherwise, it succeeds. An assertion's result can be success, nonfatal failure, or fatal failure. If a fatal failure occurs, the current function is aborted; otherwise, the program continues normally.

Google Test is integrated into the Visual Studio 2017 IDE and later versions as a default component of the Desktop development with C++ workload. This makes it easy to set up and use on Windows.[15]

We'll use Google Test in Visual Studio to test the `get_error` function shown in Listing 11-8. This function is similar to the `print_error` function, but it simply returns the error message string corresponding to the error number passed in as an argument instead of printing it.

```
char *get_error(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum) + 1;
  char* msg = malloc(size);
  if (msg != NULL) {
    errno_t status = strerror_s(msg, size, errnum);
    if (status != 0) {
      strncpy_s(msg, size, "unknown error", size-1);
    }
  }
  return msg;
}
```

Listing 11-8: The `get_error` function

Listing 11-9 shows a test for the `get_error` function. Most of the C++ code is boilerplate and can be copied without modification, including, for example, the `main` function, which invokes a function-like macro, `RUN_ALL_TESTS`, that runs your defined tests.

```
#include "pch.h"
❶ extern "C" char* get_error(errno_t errnum); // implemented in C source file

namespace {
❷ TEST(MyTestSuite, MsgTestCase) {
    EXPECT_STREQ(get_error(ENOMEM), "Not enough space");
    EXPECT_STREQ(get_error(ENOTSOCK), "Not a socket");
    EXPECT_STREQ(get_error(EPIPE), "Broken pipe");
  }
}  // namespace
```

15. To learn how to add and configure a Google Test project in Visual C++, see *https://docs.microsoft.com/en-us/visualstudio/test/how-to-use-google-test-for-cpp/*.

```
int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Listing 11-9: Unit tests for the get_error function

The two parts that aren't boilerplate are the `extern "C"` declaration ❶ and the test ❷. The `extern "C"` declaration changes the linkage requirements so that the C++ compiler linker doesn't mangle the function name, as it is wont to do. You need to add a similar declaration for each function being tested, or you can simply include the C header file within an `extern "C"` block as in the following example:

```
extern "C" {
  #include "api_to_test.h"
}
```

The test in this case uses the `TEST` macro, which defines a specific test case and takes two arguments. The first argument to the `TEST` macro is the name of the *test suite*, which is a set of test cases to be executed in a specific test cycle. The second argument is the name of the *test case*, which is a set of preconditions, inputs, actions (where applicable), expected results, and post-conditions, developed based on test conditions. For further definitions, see the International Software Testing Qualifications Board (ISTQB) at *https://glossary.istqb.org/*.

Insert any Google Test assertions, along with any additional C++ statements you wish to include, in the function body. In Listing 11-9, we used the `EXPECT_STREQ` assertion, which verifies that two strings have the same content. We used the assertion on several error numbers to verify that the function is returning the correct string for each error number. The `EXPECT_STREQ` assertion is a nonfatal assertion because testing can continue even when this specific assertion fails. This is typically preferable to fatal assertions, as it lets you detect and fix multiple bugs in a single run-edit-compile cycle. If it's not possible to continue testing after an initial failure (because a subsequent operation relies on a previous result, for example), you can use the fatal `ASSERT_STREQ` assertion.

The test case tests for several error numbers from `<errno.h>`. How many of these should be tested depends on what you're trying to accomplish. Ideally, the tests should be comprehensive, which would mean adding an assertion for every error number in `<errno.h>`. This can become tiresome, however; once you have established that your code is working, you are mostly just testing that the underlying C Standard Library functions you're using are implemented correctly. Instead, we could test the error numbers we're likely going to retrieve, but this can again become tiresome because we'd have to identify all the functions called in the program and which error codes they may return.

In Listing 11-9, we opted to implement a spot check that checks a few randomly selected error numbers from different locations in the list. The result of running this test is shown in Listing 11-10.

```
.\crash-test.exe
[==========] Running 1 test from 1 test case.
[----------] Global test environment setup.
[----------] 1 test from MyTestSuite
[ RUN      ] MyTestSuite.MsgTestCase
crash-test\TestMain.cpp(39): error: Expected equality of these values:
  get_error(128)
❶ Which is: "Unknown error"
❷ "Not a socket"
[  FAILED  ] MyTestSuite.MsgTestCase (5 ms)
[----------] 1 test from MyTestSuite (10 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (31 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] MyTestSuite.MsgTestCase

 1 FAILED TEST
```

Listing 11-10: Test run of `MyTestSuite.MsgTestCase`

Somewhat surprisingly (at least, I was surprised), two of the assertions pass, but the call to `get_error(ENOTSOCK)` fails because `strerror_s` returns `"Unknown error"` ❶ instead of `"Not a socket"` ❷.

After submitting the obligatory defect report, we're left with the problem of how to solve this issue. The first solution is to just be okay with it. This means that the user might see a vague `"Unknown error"` message instead of something more useful if this `errno` is ever returned. If you're fine with this, you can change the expected result to `"Unknown error"` or leave the test case as is, knowing you may eventually need to deploy your application with a failed test case. In either case, you should be able to determine whether the library implementer eventually repairs the reported defect.

If you're not okay with having a vague error message, you can provide a wrapper method for the `strerror_s` function that provides the missing error messages. If you plan to do this, it's now probably worth your while to perform comprehensive testing of the `get_error` and `strerror_s` functions to identify all the cases that need to be repaired.

Static Analysis

Static analysis includes any process for assessing code without executing it (ISO/IEC TS 17961:2013), with the goal of providing information about possible software defects. These analyses can be performed manually, though manual analysis rapidly becomes infeasible as program complexity increases. Instead, we can use static analysis tools.

Static analysis has practical limitations, as the correctness of software is computationally undecidable. For example, the halting theorem of computer science states that there are programs whose exact control flow cannot be determined statically. As a result, any property dependent on control flow—such as halting—may not be decidable for some programs. As a consequence, static analysis may fail to report flaws or may report flaws where they don't exist.

A failure to report a real flaw in the code is known as a *false negative*. False negatives are serious analysis errors, as they may leave you with a false sense of security. Most tools err on the side of caution and, as a result, generate false positives. A *false positive* is a test result that incorrectly indicates that a flaw is present. Tools might report some high-risk flaws and miss other flaws as an unintended consequence of trying not to overwhelm the user with false positives. False positives can also occur when the code is too complex to perform a complete analysis. The use of function pointers and libraries can make false positives more likely.

Ideally, tools are both complete and sound in their analysis. An analyzer is considered *sound* if it cannot give a false-negative result. An analyzer is considered *complete* if it cannot issue false positives. The possibilities for a given rule are outlined in Figure 11-1.

		False Positive	
		Yes	No
False Negative	Yes	Incomplete and unsound	Complete and unsound
	No	Incomplete and sound	Complete and sound

Figure 11-1: Completeness and soundness

The compilation process performs a limited amount of static analysis, providing diagnostics about highly localized issues in code that don't require much reasoning. For example, when comparing a signed value to an unsigned value, the compiler may issue a diagnostic about a type mismatch because it doesn't require additional information to identify the error. The flags we discussed earlier, such as `/W4` for Visual C++ and `-Wall` for GCC and Clang, control the compiler output you see.

Compilers generally provide high-quality diagnostics, and you shouldn't ignore them. Always try to understand the reason for the warning and rewrite the code to eliminate the error, rather than simply quieting warnings by adding type casts or making arbitrary changes until the warning goes away. See the CERT C rule MSC00-C (Compile cleanly at high warning levels) for more information on this topic.

Once you've addressed compiler warnings in your code, you can use a separate static analyzer to identify additional flaws. Static analyzers will diagnose more-complex defects by evaluating the expressions in your program, performing in-depth control and data flow analysis, and reasoning about the possible ranges of values and control flow paths taken.

Having a tool locate and identify specific errors in your program is much, much easier than hours of testing and debugging, and much less costly than deploying defective code. A wide variety of free and commercial static analysis tools are available. For example, Visual C++ has static analysis capabilities that you can enable with the `/analyze` flag. Visual C++ analysis allows you to specify which rule sets (such as recommended, security, or internationalization) you would like to run, or whether to run them all. For more information on Visual C++'s static code analysis, see Microsoft's website at *https://docs.microsoft.com/en-us/visualstudio/code-quality/code-analysis-for-c-cpp-overview/*. Similarly, Clang has a static analyzer that can be run as a standalone tool or within Xcode (*https://clang-analyzer.llvm.org/*). GCC 10 will also have a simple static analyzer. Commercial tools also exist, such as CodeSonar from GrammaTech, TrustInSoft Analyzer, SonarQube from SonarSource, Coverity from Synopsys, LDRA Testbed, Helix QAC from Perforce, and others.

Many static analysis tools have nonoverlapping capabilities, so it may make sense to use more than one.

Dynamic Analysis

Dynamic analysis is the process of evaluating a system or component during execution. It's also referred to as *runtime analysis*, among other similar names.

A common approach to dynamic analysis is to *instrument* the code—for example, by enabling compile-time flags that inject extra instructions into the executable—and then run the instrumented executable. A similar approach is taken by the debug memory allocation library `dmalloc` described in Chapter 6. The `dmalloc` library provides replacement memory management routines with runtime-configurable debugging facilities. You can control the behavior of these routines by using a command line utility (also called `dmalloc`) to detect memory leaks, and to discover and report defects such as writing outside the bounds of an object and using a pointer after it's been freed.

The advantage of dynamic analysis is that is has a low false-positive rate, so if one of these tools flags a problem, fix it!

A drawback of dynamic analysis is that it requires sufficient code coverage. If a defective code path is not exercised during the testing process, the defect won't be found. Another drawback is that the instrumentation may change other aspects of the program in undesirable ways, such as adding performance overhead or increasing the binary size.

AddressSanitizer is an example of an effective dynamic analysis tool that is available (for free) for several compilers. Several related sanitizers exist, including ThreadSanitizer, MemorySanitizer, Hardware-Assisted AddressSanitizer, and UndefinedBehaviorSanitizer.[16] Many other dynamic analysis tools are available, both commercial and free. I'll demonstrate the value of these tools by discussing AddressSanitizer in some detail.

16. For more information on sanitizers, see *https://github.com/google/sanitizers/*.

AddressSanitizer

AddressSanitizer (ASan) is a dynamic memory error detector for C and C++ language programs; see *https://github.com/google/sanitizers/wiki/AddressSanitizer*. It's incorporated into LLVM version 3.1 and GCC version 4.8, as well as later versions of these compilers. ASan is also available starting with Visual Studio 2019. This dynamic analysis tool can find a variety of memory errors, including the following:

- Use after free (dangling pointer dereference)
- Heap, stack, and global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

To demonstrate ASan's usefulness, we'll instrument the rewritten `print_error` function from Listing 11-7 and the `get_error` function from Listing 11-8 using ASan, and then analyze the code on Ubuntu Linux. We've already developed unit tests for the `get_error` function from Listing 11-9, which we'll extend for this purpose.

The Google Test code in Listing 11-11 tests the two utility functions for printing errors. In addition to the tests ensuring that the `get_error` function returns the correct string, we've added the nonfatal `EXPECT_EQ` assertion that tests for a return value of 0 from the call to `print_error`.

```
TEST(PrintTests, MsgTestCase) {
  ASSERT_STREQ(get_error(ENOMEM), "Not enough space");
  ASSERT_STREQ(get_error(ENOTSOCK), "Not a socket");
  ASSERT_STREQ(get_error(EPIPE), "Broken pipe");
  EXPECT_EQ(print_error(ENOMEM), 0);
  EXPECT_EQ(print_error(ENOTSOCK), 0);
  EXPECT_EQ(print_error(EPIPE), 0);
}
```

Listing 11-11: Error print tests

Next, we need to build and run this code on Ubuntu Linux.

Building the Code on Ubuntu Linux

If you managed to compile and test the `get_error` function from Listing 11-8, you should have a working bounds-checking interface implementation on your Ubuntu Linux machine. Next, you may need to install the Google Test development package by using the `apt-get` command on Ubuntu:

```
% sudo apt-get install libgtest-dev
```

This package installs only source files, so you'll need to compile the code to create the necessary library files. Change directories to the */usr/src/gtest* folder that should house these source files and use `cmake` to compile the library, using the following commands:

```
% sudo apt-get install cmake # install cmake
% cd /usr/src/gtest
% sudo cmake CMakeLists.txt
% sudo make
% # copy or symlink libgtest.a and libgtest_main.a to your /usr/lib folder
% sudo cp *.a /usr/lib
```

You should now be able to build and run `PrintTests` from Listing 11-11 on your Ubuntu machine before continuing. This will likely require modifications to the source code and build files.

Running the Tests

The `strerror_s` function used by the `get_error` and `print_error` functions returns a locale-specific message string. Run the tests from Listing 11-11, and you'll probably notice that something is not quite right—namely, the tests for the `get_error` function all fail. This is because these tests were originally developed using Visual C++ for Windows, and Ubuntu Linux returns different locale-specific message strings. If this wasn't the behavior you expected, you may need to rewrite these two functions so that they return the same string regardless of locale. Otherwise, we can rewrite the tests as shown in Listing 11-12 to verify that the anticipated locale-specific message strings (shown in bold font) are returned.

```
TEST(PrintTests, MsgTestCase) {
  EXPECT_STREQ(get_error(ENOMEM), "Cannot allocate memory");
  EXPECT_STREQ(get_error(ENOTSOCK), "Socket operation on non-socket");
  EXPECT_STREQ(get_error(EPIPE), "Broken pipe");
  EXPECT_EQ(print_error(ENOMEM), 0);
  EXPECT_EQ(print_error(ENOTSOCK), 0);
  EXPECT_EQ(print_error(EPIPE), 0);
}
```

Listing 11-12: Revised error print tests

Running the revised tests from Listing 11-12 should now produce the positive results shown in Listing 11-13. An inexperienced tester may look at these results and mistakenly think, "Hey, this code is working!" However, you should take additional steps to improve your confidence that your code is free from defects.

```
student@scode:~/Examples/asan$ ./runTests
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from PrintTests
[ RUN      ] PrintTests.MsgTestCase
```

```
Cannot allocate memory
Socket operation on non-socket
Broken pipe
[       OK ]
PrintTests.MsgTestCase (0 ms)
[----------] 1 test from PrintTests (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

Listing 11-13: Test run of `PrintTests`

Now that we have a working test harness in place, it's time to instrument the code.

Instrumenting the Code

You can instrument your code by using AddressSanitizer to compile and link your program with the `-fsanitize=address` flag. To get more-informative stack traces in error messages, add the `-fno-omit-frame-pointer` flag, and to get symbolic debugging information, add the `-g3` flag. Using `cmake`, you can add these flags by including the following line in your *CMakeLists.txt* file:

```
set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -g3 -fno-omit-frame-pointer -fsanitize=address")
```

As previously mentioned, AddressSanitizer works with Clang, GCC, and Visual C++.[17] Depending on which version of which compiler you're using, you may also need to define the following environment variables:

```
ASAN_OPTIONS=symbolize=1
ASAN_SYMBOLIZER_PATH=/path/to/llvm_build/bin/llvm-symbolizer
```

With these set, try rebuilding and rerunning your tests.

Running the Tests

The unit tests you wrote using Google Test should continue to pass but will also exercise your code by allowing AddressSanitizer to detect additional problems. You should now see the additional output in Listing 11-14 from running `PrintTests`.

```
❶ ==16447==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 31 byte(s) in 1 object(s) allocated from:
    #0 0x7fd8e3a1db50 in __interceptor_malloc
    (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xdeb50)
  ❷ #1 0x564622aa0b39 in print_error ~/asan/PrintUtils/print_utils.c:12
```

17. See "AdressSanitizer (ASan) for Windows with MVSC" at the Microsoft C++ Team Blog, *https://devblogs.microsoft.com/cppblog/addresssanitizer-asan-for-windows-with-msvc/*.

```
    #2 0x564622a65839 in TestBody ~/asan/TestMain.cpp:49
    #3 0x564622a91754 in void
       testing::internal::HandleSehExceptionsInMethodIfSupported
       <testing::Test, void>(testing::Test*, void (testing::Test::*)(),
       char const*) (~/asan/runTests+0x35754)
    #4 0x564622a8b75c in void
       testing::internal::HandleExceptionsInMethodIfSupported
       <testing::Test, void>(testing::Test*, void (testing::Test::*)(),
       char const*) (~/asan/runTests+0x2f75c)
    #5 0x564622a6f139 in testing::Test::Run() (~/asan/runTests+0x13139)
    #6 0x564622a6fa6f in testing::TestInfo::Run() (~/asan/runTests+0x13a6f)
    #7 0x564622a700f9 in testing::TestCase::Run() (~/asan/runTests+0x140f9)
    #8 0x564622a76fc5 in testing::internal::UnitTestImpl::RunAllTests()
       (~/asan/runTests+0x1afc5)
    #9 0x564622a9291a in bool
       testing::internal::HandleSehExceptionsInMethodIfSupported
       <testing::internal::UnitTestImpl, bool>(testing::internal::UnitTest
       Impl*, bool (testing::internal::UnitTestImpl::*)(), char const*)
       (~/asan/runTests+0x3691a)
    #10 0x564622a8c598 in bool
        testing::internal::HandleExceptionsInMethodIfSupported
        <testing::internal::UnitTestImpl, bool>(testing::internal::UnitTest
        Impl*, bool (testing::internal::UnitTestImpl::*)(), char const*)
        (~/asan/runTests+0x30598)
    #11 0x564622a75b89 in testing::UnitTest::Run() (~/asan/runTests+0x19b89)
    #12 0x564622a66715 in RUN_ALL_TESTS() /usr/include/gtest/gtest.h:2233
    #13 0x564622a65fd7 in main ~/asan/TestMain.cpp:57
    #14 0x7fd8e2b13b96 in __libc_start_main
        (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
```

Listing 11-14: Instrumented test run of `PrintTests`

Listing 11-14 shows only the first finding of several that are produced. Most of this stack trace is from the test infrastructure itself and is uninteresting because it doesn't help locate the defects. All of the interesting information is at the top of the listing (and the stack).

First, we're told that `LeakSanitizer`, a component of AddressSanitizer, has "detected memory leaks" ❶ and that this is a direct leak of 31 bytes from one object. The stack trace refers us to the following line of code ❷:

```
#1 0x564622aa0b39 in print_error ~/asan/PrintUtils/print_utils.c:12
```

This line of code contains the call to `malloc` in the `print_error` function:

```
errno_t print_error(errno_t errnum) {
  rsize_t size = strerrorlen_s(errnum);
  char* msg = malloc(size);
  //---snip---
}
```

This is a fairly obvious error; the return value from `malloc` is assigned to an automatic variable defined within the scope of the `print_error` function and never freed. We lose the opportunity to free this allocated memory

after the function returns and the lifetime of the object holding the pointer to the allocated memory ends. To fix this problem, add a call to `free(msg)` after the allocated storage is no longer required but before the function returns. Rerun the tests and repair any detected defects until you're satisfied with the quality of your program. If you're feeling particularly sadistic, deploy your code on a Friday and take your phone off the hook for the weekend.

Exercises

Try these code exercises on your own:

- Use the static analyzer built into Visual C++ to evaluate the defective code from Listing 11-1. Did the static analysis provide any additional findings?
- Evaluate the remaining results from the `PrintTests` test instrumented with AddressSanitizer. Eliminate the remaining true-positive errors detected.
- Try instrumenting the `PrintTests` test using other sanitizers available at *https://github.com/google/sanitizers/* and address any issues you find.
- Use these and similar testing, debugging, and analysis techniques on your real-world code.

Summary

In this chapter, you learned about static and runtime assertions and were introduced to some of the more important and recommended compiler flags for GCC, Clang, and Visual C++. You also learned how to debug, test, and analyze your code by using both static and dynamic analysis. These are important last lessons in this book, because you'll find you spend a considerable amount of time as a professional C programmer debugging and analyzing your code.