Programming in C

Rex Jaeschke

Programming in C

Edition: 5.0

The training materials associated with this book are available for license. Interested parties should contact the author.

Please address comments, corrections, and questions to the author, Rex Jaeschke, at rex@RexJaeschke.com.

Much of the information in this book has been published previously, as follows:

1. A series of monthly columns called "Let's C Now" in *DEC Professional*, Professional Press, running from 1984–1995.
2. A two-volume set of reprints called "Let's C Now", Professional Press, 1986 and 1987, ISBN 0-9614729-2-8 and 0-9614729-3-6.
3. The book "Mastering Standard C", Rex Jaeschke, Professional Press 1989, ISBN 0-9614729-8-7.
4. The book "Mastering Standard C", Rex Jaeschke, 2e, CBM Books 1996, ISBN 1-878956-55-8.

The glossary of terms from "Mastering Standard C", 1e, was spun-off and expanded into its own book, "The Dictionary of Standard C", Prentice Hall 2001, ISBN 0-13-090620-4. This dictionary can be viewed on-line, free of charge, at www.prenhall.com/jaeschke. An earlier version of this dictionary, now out of print, was published by Professional Press Books 1991, ISBN 0-07-707657-5.

Preface

Welcome to the world of C. Throughout this book, we look at the statements and constructs of the C programming language. Each statement and construct is introduced by example with corresponding explanations, and, except where errors are intentional, the examples are complete programs or subroutines that are error-free. I encourage you to run and modify these examples, because the only realistic way to master a language is to write programs in it, run them, and debug them.

This book was written with teaching in mind. It is intended for use both in a classroom environment as well as for self-paced learning.

Dennis Ritchie developed C around 1972 at AT&T's Bell Laboratories. C evolved from the languages CPL, BCPL, and B, in that order. The UNIX operating system, another Bell Labs development (designed by Ken Thompson and Ritchie), then was rewritten in C. (Previously, UNIX was written in assembler and B.) UNIX and C have been closely associated ever since, and every UNIX and UNIX -like operating system includes a C compiler. Today, C has a life quite separate from UNIX, with implementations of C available for every mainstream hardware and operating system platform.

C is a general-purpose, high-level language. From a grammatical viewpoint, C is a relatively simple language. Because of the power of the basic statements and constructs, and the fact that the programmer can effectively extend the language by using callable functions while still maintaining portability, there is little need for language extensions especially considering the fact that C supports calls to procedures written in other languages. However, extensions do exist in some compilers and vary from one vendor to another. You should be aware of any nonstandard conventions in production compilers you use or evaluate. Most people learn a language by using one particular dialect of it. As a result, they are often unaware when they are using an extension.

Depending on their language backgrounds, programmers new to C may initially find programs hard to read because, traditionally, most C code is written in lowercase. Lower- and uppercase letters are treated by the compiler as distinct. In fact, most C language keywords must be written entirely in lowercase.[1] Keywords are also reserved words.

C supports a structured, modular approach to programming using callable subroutines, called *functions*. Source files can be compiled separately, with external references being resolved at linktime.

C lends itself to writing terse code. However, there is a fine (and subjective) line between writing code that is terse and code that is cryptic. It is easy to write code that is unreadable and, therefore, unmaintainable. However, with care and a small dose of discipline, you can produce nicely formatted code that is easy to read and maintain. However, good code doesn't happen automatically—you must work at it. Throughout the book, I make numerous comments and suggestions regarding style. Perhaps the best advice I can give in that regard is "Remember that the poor fool who has to read your code in the future just might be you!" Above all, be overt and be consistent.

---

[1] Starting with C99, new kewords are spelled with a leading underscore followed by an uppercase letter, followed by lowercase letters and underscores, as in `_Bool` and `_Thread_local`.

Numerous "Tips" and "Style Tips" have been added throughout to highlight important suggestions. Considerable attention has been paid to using and teaching a consistent, popular, and overt programming style. Both kinds of tips are highlighted so they stand out. For example:

> **Tip:** Avoid initializing enumeration constants explicitly or making duplicates unless you have good reason to do so. Assigning a range of values that is non-continuous or that contains duplicates significantly reduces the set of operations that can meaningfully be performed.

> **Style Tip:** Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it. C is not all things to all people; nor does it claim to be. For many applications, assembly language, Pascal, COBOL, FORTRAN, and BASIC, for example, will do just fine. In any event, compared to other high-level languages (including Java and C#), C is a relatively expensive language to learn and master.

Reader Assumptions

This is *not* a first course in programming. (Nor do I recommend C as a first programming language.)

I assume that you know how to use your text editor, C compiler, and debugger. Comments on the use of these utility programs will be limited to points of interest to the C programmer.

To fully understand and exploit the material, you should be conversant with the following topics:

- The basic purpose of a compiler and linker.
- Number system theory.
- Bit operations such as AND, inclusive OR, exclusive-OR, complement, and left- and right-shift.
- Data representation.
- Communication between procedures by passing arguments and/or by returning a value, as well as via global variables.
- Use of single- and multi-dimensional arrays.
- Creation and use of sequential files and how to do formatted and unformatted I/O.
- Basic data structures such as linked lists.

If your programming background is in some procedural language such as PL/I, Pascal, FORTRAN, or BASIC, you'll need to read the whole book closely and do all the exercises. As for those of you having only a COBOL background, you'll have some catching-up to do with respect to thinking in binary versus decimal when it comes to data representation, as well as the use of functions and their associated argument passing and value returning machinery. If you are coming from scripting languages, Java, or C#, many things look familiar; however, C's pointer syntax and preprocessor use will be different.

Limitations

This book covers almost all the C language. It also introduces the core class library. However, a very small percentage of library facilities are mentioned or covered in any detail. The Standard C library contains so many functions that whole books have been written about that subject alone.

This book is directed at teaching the C language proper, by writing simple stand-alone applications, since without a thorough knowledge of the language, you will not be able to understand and implement other, more advanced, solutions.

GUI, calling non-C routines, threading, inter-process communications, operating system-specific, and many other advanced features are outside the scope of this text.

A companion volume, “Advanced Programming in C”, includes the following topics, among others: pointers to functions, pointers to arrays, the comma operator, sequence points, lvalues, internationalization, threads, and numerous advanced library topics.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming language training for more than 20 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and focused. Many books use GUI and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than is any other; I simply know that my approach works well, and has formed the basis of my successful seminar business.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory, within which each program may well have its own subdirectory. For example, the source code for the program called ba04 in the "Basics" chapter can be found in the following directory hierarchy: Source, Basics, ba04. By convention, the names of C source files end in “.c”.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided electronically in a directory tree named Labs, where each chapter has its own subdirectory, within which each program has its own subdirectory.[1] For example, lab solution lbba01 in the "Basics" chapter has the following fully qualified directory hierarchy: Labs, Basics, lbba01.

Exercises that are not so marked have no general solution and may require experimentation or research in an implementation's documentation.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

Program Behavior

According to the C Standard, for correct, well-formed programs, almost all behaviors are well defined and predictable. However, in certain cases, an implementation can choose the behavior it deems best, and this behavior need not be the same as that exhibited by other implementations. Such behavior is called *implementation-defined*, and must be documented by each implementation. For example:

> **Implementation-Defined Behavior:** The range of values that can be stored in an object of a given arithmetic type.

In other cases, an implementation can choose whatever behavior it wants at that time *without* needing to reproduce or document that choice. Such behavior is called *unspecified*. For example:

> **Unspecified Behavior:** Whether two string literals containing the same characters are stored in distinct arrays.

A third category is *undefined behavior*, which can result from situations for which the standard imposes no requirements or is otherwise silent. For example:

> **Undefined Behavior:** Subscripting an array with an out-of-bounds index.

Clearly, we must be aware of implementation-defined and unspecified behaviors when writing code that is to be ported across different platforms. And we should always avoid relying on undefined behavior.

Far too many people believe that just because their C code has worked for a long time, it must be correct. When next your compiler is upgraded, the behavior it exhibits with "undefined behavior" constructs may well be different (as it is permitted to be), resulting in old code breaking. In some cases, simply using a different combination of compiler options with the same compiler can change the way the compiler works internally. You should never try to figure out how your compiler works in "undefined" cases. Any test case you write will be so trivial as to be meaningless. And having 10, 100, or even 1,000 test cases exhibit the same "undefined" behavior is still no guarantee that the next test won't behave differently.

The Status of Standard C

The history of the standardization of C is as follows:

- C89 – The first ANSI C standard, ANSI X3.159-1989, was produced in 1989 by the U.S. committee X3J11.
- C90 – The first ISO C standard, ISO/IEC 9899:1990, was produced in 1990 by committee ISO/IEC JTC 1/SC 22/WG 14 in conjunction with committee X3J11. C90 was technically equivalent to C89.
- C95 – An amendment to C90 was produced in 1995 by committee WG 14 in conjunction with the U.S. committee X3J11. The additions included digraphs, the header `iso646.h`, and many multibyte and wide-character functions via the headers `wchar.h` and `wctype.h`.
- C99 – The second edition of the ISO C standard, ISO/IEC 9899:1999, was produced by committee WG14 in conjunction with the U.S. committee INCITS/J11 (formerly X3J11). The additions included a few language features, a number of headers, and many library functions. Throughout its development, C99 was commonly referred to as C9x.

- C11 – The third edition of the ISO C standard, ISO/IEC 9899:2011, was produced by committee WG14 in conjunction with the U.S. committee INCITS/PL22.11 (formerly INCITS/J11). The additions included support for multiple threads of execution, processing Unicode characters and strings, and the querying and specification of the alignment of objects, among other things.
- C17 – The fourth edition of the ISO C standard, ISO/IEC 9899:2017, was produced by committee WG14 in conjunction with the U.S. committee INCITS/PL22.11. This was a maintenance release that included corrections to Defect Reports. No new functionality was added.

The C standards committee is currently working on various maintenance issues and informative Technical Reports.

Electronic copies of the latest C standard can be purchased from www.ansi.org or www.iso.ch.

Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my C seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, June 2018

1. The Basics

In this chapter, we will learn about a number of fundamental constructs and language elements.

1.1 Basic Program Structure

A correct source program is made up of an acceptable combination of tokens, where a *token* is the smallest possible unit of source. C has five different kinds of tokens: keywords, identifiers, literals, operators, and punctuators.

For the most part, C is a free-format language, with space between tokens being optional. However, in some cases, some kind of separator is needed between tokens, so they can be recognized the way they were intended. White space performs this function. *White space* consists of one or more consecutive characters from the following set: space, horizontal tab, vertical tab, form-feed, and *new-line* (entered by pressing the RETURN or ENTER key). In a source file, an arbitrary amount of white space is permitted before the first token, between any two adjacent tokens, and after the last token.

Let's look at the basic structure of a C program, which writes a welcome message to the console (see directory ba01):

```
/*–
This program is a simple example of C.
–*/

/*1*/ #include <stdio.h>

int main()              /* start of the program */
{                       /* beginning of function body */

/*2*/   printf("Welcome to C\n");

/*3*/   return 0;       /* terminate with success */
}                       /* end of a function body */
```

A delimited comment is one surrounded by `/*` and `*/`, and can be used on part of a source line, as a whole source line, or it can span any number of source lines. C99 added C++'s line-oriented comments, which begin with `//` and continue until the end of the same source line. A comment of either kind is treated as a single space. A delimited comment can occur anywhere white space can be used. A line-oriented comment can only appear at the end of a source line. Although comments of the same kind do not nest, each kind of comment can be used to disable a source line containing the other.[1]

A C program consists of one or more functions that can be defined in any order in one or more source files (or *translation units*, as Standard C calls them). A program must contain at least one function, called `main`, and this

[1] Refer to §6.4 to see a novel way to "comment out" a source line containing a comment.

function's name must be spelled using lowercase letters.[1] This specially named function indicates where the program is to begin execution.[2] The `int` keyword preceeding `main` indicates that `main` will return a result of type `int`, as discussed below.

The parentheses following the function name `main` surround the function's parameter list.[3] The parentheses are required, even if no arguments are expected.

The body of a function is enclosed within a matching pair of braces. All executable code must reside within the body of some function or other. Statements are executed in sequential order unless branching or looping statements dictate otherwise. A program terminates when it returns from `main`.

We'll learn about the `return` statement in §4.3. However, why return a value from `main`? Many programs either are invoked at the operating system's command-line level or are spawned by another program. In either case, when a program terminates, it has the ability to return one piece of information to the program that invoked it. Perhaps it returns a status code or a count of the number of transactions processed. We refer to this returned value as the program's *exit status code*. The value used, and its meaning are the business of the programmer. Unless otherwise stated, all `main` functions in this book contain the statement `return 0;`. The value zero has no special significance, although on some systems it is interpreted as some form of success code.

Case 1[4] provides access to the I/O library, and case 2 outputs the welcome text to the console. These cases are discussed in detail in §1.3.

> **Style Tip:** Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. **If you can't read the code, you surely won't be able to understand it.**

We can write any given correct program in an infinite number of ways simply by using different amounts and kinds of white space (including comments). Obviously, there are very good, overt styles and very bad, cryptic styles. Then there is the large and subjective gray area in between.

> **Style Tip:** Be overt; pick a style that isn't too far outside the mainstream, and stick with it. Above all, be consistent. And remember, the style you develop when learning a language is the one with which you will likely stay, so give some thought to programming style from the outset.

The style used in this book is very common. If you choose to adopt a different style, note that no matter how great you think your style is, every minute you spend arguing about whose style is superior and why—or worse yet, changing other people's code to your style—is generally time wasted. If no corporate-wide or group style guide exists, and the people on a multi-person project cannot agree on a common style, the project manager should dictate one.

---

[1] While mixed- or uppercase spellings of `main` might be recognized by some systems, this is undefined behavior.
[2] Some environments require a different entry-point name. For example, certain Microsoft Windows programs begin execution at `winmain` instead. Likewise, for programs that run on an embedded system.
[3] A function uses parameters to declare what it is expecting to be passed to it, via an argument list, at run time.
[4] Throughout this book, many code examples contain source lines with leading delimited comments of the form `/*`*n*`*/`, where *n* is a number. Such lines are referred to as "cases", with `/*1*/` being case 1, `/*2*/` being case 2, and so on.

Exercise 1-1*: Compile and link a program whose `main` contains only a `return` statement, and look at the size of the resulting executable file on disk; it may be surprisingly large. Look at the listing produced by the linker and see what goes into a seemingly empty program. (See lab directory lbba03.)

Exercise 1-2*: Try compiling, linking, and executing a main program whose name is something like `Main` or `MAIN`. If either of these works, use `xyz` instead and try again. (See lab directory lbba04.)

Exercise 1-3*: Using `/*` and `*/`, comment out a block of code that already contains this kind of comment, and see how your compiler reacts. (Some compilers actually do support nested comments as an extension.) (See lab directory lbba05.)

Exercise 1-4*: What happens if you leave off the closing `*/` from a comment? (See lab directory lbba06.)

Let's move on to the more common case of a program having multiple functions (see directory ba02):

```
/* C program with two functions */

void compute()
{
        /* … */
/*1*/   return;         /* redundant statement */
}

int main()
{
/*2*/   compute();      /* call function 'compute' */

        return 0;
}
```

C supports modularization via *functions*. A source file can contain one or more functions, defined in any order. (That said, in this example, it is no accident that function `compute` is defined before function `main`. In §4, we'll learn how to define functions in any order as well as how to put them in separate source files.) Unlike some languages, in C, function definitions cannot be nested. That is, each function's definition must be outside the braces delimiting the definitions of all other functions. There is no syntactic difference between `main` and any other C function; they all have the same basic structure.

We invoke (that is, *call*) a function by using its name followed by a possibly empty argument list, as shown in case 2. The parentheses used in the call represent the function call operator.

Each statement must be terminated by a semicolon punctuator. When function `compute` terminates, control is returned to its caller. The explicit `return` statement in case 1, is redundant; dropping into the closing brace of a function is an implicit `return` without a value (The presence of the keyword `void` indicates that `compute` does not return any value.)

Style Tip: Indent every line inside the body of a function definition by at least one tab. Also, line up the opening and closing braces, one above the other, but not indented with the block they delimit. This way, you can see the shape of the program at a glance by looking at the brace pairs. And if you are interested in the details, you can quite easily see where they are.

Exercise 1-5: See how your compiler reacts to having one function defined inside another. (This can easily happen if you forget the closing brace from the first function. And if you don't line up your brace pairs, you won't easily see which one is missing.)

Exercise 1-6*: What happens when you omit the semicolon from the end of a statement? Add an extra one and see what happens. (See lab directory lbba07.)

1.2 Identifiers

One kind of source token is an *identifier*, a name usually invented by the application programmer. Identifiers are used to name variables and functions, among other things. Since keywords (such as `int`, `void`, and `return`) are reserved, they cannot be used as identifiers.

An identifier can be spelled using the following characters: Upper- and lowercase Latin letters (which are distinct), the decimal digits 0–9, and the underscore.[1] However, an identifier cannot begin with a digit, and identifiers that begin with an underscore followed by a capital letter, or two underscores, are reserved for use by C implementers. (§6.6 show several ways in which an implementation might provide names having a reserved spelling format.)

Style Tip: To avoid possible conflict with "private" names used by your implementation, never invent an identifier that begins with an underscore or that contains two consecutive underscores.

An identifier can be arbitrarily long.

Implementation-Defined Behavior: The number of characters in an identifier that are treated as significant.[2]

While we can spell identifiers using letters in either case, or combinations of upper- and lowercase, the most common styles of naming variables and functions use either all lowercase or mostly lowercase with uppercase leading characters. Examples are `total`, `maximumNumber`, and `Week_Day`. Identifiers spelled in all uppercase are often used for other purposes, as we shall see in §6.

[1] Some implementations also permit identifiers to contain `$` or other characters, as an extension; however, the use of such characters in a program makes it non-portable. The Advanced-C book shows how to include non-English letters via Universal Character Names.

[2] Modern compilers support at least 31 significant characters.

Style Tip: It is very bad style to invent an identifier such as `If`, `Else`, or `FOR`, since such names are too easily confused by the reader (but never by the compiler) with keywords having the same names but spelled in lowercase.

Exercise 1-7*: Which of the following are valid identifiers: `name`, `_Xyz_`, `_`, `Total.count`, `Today'sdate`, `first-name`, `day_of_week`, `X32BITS`, `TOTAL$COST`, `3WiseMen`, `LastNameOfMyFathersGrandfatherInLaw`? (See lab directory lbba02.)

1.3 Introduction to Formatted I/O

Unlike many older languages, C has no input or output statements. Instead, a set of I/O functions is provided as part of the standard library. One such library function is `printf`. This function permits formatted output to be written to the *standard output* device, which typically is directed to the user's terminal. For example (see directory ba03a):

```
/* write to standard output */

#include <stdio.h>

int main()
{
/*1*/   printf("Hello.\n");
/*2*/   printf("\nWelcome");
/*3*/   printf(" to C.\n");

/*4*/   printf("A few \
words\n");

/*5*/   printf("Three "
               "separate "
               "words\n");
}
```

The output produced is:

```
Hello.

Welcome to C.
A few words
Three separate words
```

Prior to calling a standard library routine, we must tell the compiler how that function should be called. This allows the compiler to check that the number and type of arguments passed correspond to those expected. In the case of `printf`, we do this by using `#include <stdio.h>`. This is a directive aimed at the *C preprocessor*, a

program that processes the source before passing it off to the compiler proper.[1] In this directive, the standard *header* called `stdio.h` is included.

Headers should be included at the beginning of a source file, prior to any function definitions.

We can think of a header as a source file that contains shareable information. In this case, the header `stdio.h` contains declarations for the standard I/O library functions. The compiler uses this information to check calls to those library routines.

In case 1, `printf` is called with one argument, a *string literal*. The characters enclosed in double quotes are printed verbatim, except for certain *escape sequences*, which begin with a backslash. \n is C's notation for a *new-line*, the character that moves the print position to the first character on the next output line. A new-line is not automatically appended by `printf`, so `printf` can be invoked multiple times to print an output line a piece at a time, as shown in cases 2 and 3. `printf` can also output multiple lines at a time. If we want double spacing, we must use two consecutive new-line escape sequences.

In cases where we have long string literals, or our style requires considerable indenting, it may be necessary to break a string literal over multiple lines. Now while we can put an arbitrary amount of white space between two adjacent tokens, we cannot split a single token, such as a string literal. To handle this case, we can break the string literal anywhere we like by using a backslash, and continue it on the next line, as shown in case 4. No white space—not even comments—is permitted after the backslash terminator, and we must continue the string in the first position of the next source line. The compiler will splice the line containing the backslash terminator to the source line immediately following it.

Requiring continuation in column 1 impedes readability, however, and an alternative approach was invented, as shown in case 5. This involves breaking the string into a number of smaller strings, separating them with white space as desired. The compiler concatenates adjacent string literal tokens.

Inside string literals, the backslash is used as an escape character prefix. It indicates that the following one or more characters are to be interpreted with other than their literal meaning. Each escape sequence represents a single character. The complete set of escape sequences is:

Table 1-1: Escape Sequences

Sequence	Meaning
\a	Alert
\b	Backspace
\f	Form feed
\n	Line-feed
\r	Carriage return
\t	Horizontal tab

[1] Preprocessor directives are discussed in detail in 6.

Sequence	Meaning
\v	Vertical tab
\\	Backslash
\'	Single quote
\"	Double quote
\?	Question mark
ooo	Octal bit pattern *ooo*
\x*h..h*	Hexadecimal bit pattern *h..h*

Undefined Behavior: If any escape sequence other than those shown in the table above is encountered.[1]

For the most part, the escape sequence mechanism allows character-set independence for these special characters.[2] Most are self-explanatory; however, a few require further explanation. Alert sounds the terminal bell, if one exists. Carriage return is a holdover from the early days. It is unlikely it will ever be needed; a new-line will always suffice as a line terminator. Sometimes it is necessary to escape a single quote, as we shall see in §1.5.2. To enclose a double quote in a string literal, we must use \". The question mark sequence is primarily of interest to programmers in countries (such as those in Scandinavia) where terminals using the ISO-646 character set are prevalent.

The octal and hexadecimal bit pattern sequences allow special commands to be sent to output devices. For example, to clear the screen of a terminal that understands ANSI hardware escape sequences, we print the character sequence `\33[2J`, where `\33` represents the ASCII ESCAPE character. Likewise, in the olden days, we could use escape sequences to set a dot-matrix printer to bold, italic, or underline mode.

`printf` can output the value of variables as well as text and escape sequences, as the next example demonstrates (See directory ba03b):

```
#include <stdio.h>

int main()
{
/*1*/   int year;
/*2*/   int i = 10;     /* Use an initializer */
```

[1] Some implementations define extra escape sequences as extensions.
[2] The Advanced-C book shows how to represent non-ASCII characters using an escape-sequence-like notation.


```
/*3*/   year = 1992;
/*4*/   printf("The year %d was a leap year.\n", year);

/*5*/   printf("%d squared = %d\n", i, i * i);

        return 0;
}
```

The output produced is:

```
The year 1992 was a leap year.
10 squared = 100
```

The variables `year` and `i` are defined to be of type `int`, a signed integer type that can represent a range of values that includes 1992. (We will discuss the range of each integer type in §1.4.1.)

Variables must be declared explicitly before their first use. Unlike some languages, when an unknown variable name is used, there is no implicit creation or typing; it is diagnosed as an error. Note that, like statements, each declaration must be terminated by a semicolon.

The terms *definition* and *declaration* are quite often used interchangeably, even though there is a subtle difference. Therefore, we should be careful to use them correctly. A declaration of an identifier declares the attributes of that identifier. In the case of `year`, it declares that `year` is an object of type `int`. A definition not only declares an identifier, it also allocates memory for it. Every definition is also a declaration; however, not every declaration is a definition, as we shall see in §4.7 and §9.

We assign values to variables using the = assignment operator. The value assigned can be a constant, the value of another variable, the result returned from a function, or the value of any expression having *compatible type*.[1] If necessary, the type of the result of the right-hand expression is converted to match that of the left-hand expression. In case 2, the definition of `i` includes the *initializer* = `10`, eliminating the need for a separate assignment statement for that variable.

In case 4, `printf` is called with two arguments: a string literal and the integer expression `year`. The string literal now includes display format information via a *conversion specifier*, a special sequence beginning with `%`. `%d` tells `printf` that the next argument has type `int` and that its value is to be displayed as a decimal integer using only the number of print positions needed. A leading sign will be printed only if the value is negative.

We can see from the output that the `%d` is replaced by the value of the second argument. Conversion specifiers exist for each arithmetic data type, and the specifiers present in the string determine the number, order, and type of arguments `printf` should expect after the format string. We will introduce other conversion specifiers in future examples. (The complete set of conversion specifiers is listed in §D.1.)

> **Undefined Behavior:** If the number and/or type of arguments following the format string in a call to `printf` do not match those specified in the format string.

[1] All arithmetic types are compatible with one another.

However, it is permitted to specify fewer conversion specifiers than there are arguments, in which case, the excess arguments are evaluated, but are otherwise ignored. One down side to I/O's not being part of the language is that the compiler is not required to compare the conversion specifiers with the number and types of the arguments that follow. The responsibility for this matching is left solely to the programmer; if you lie to `printf`, that's *your* problem:

In case 5, `printf` has three arguments, the last of which is an expression with the value 100. The value of each of the two arguments after the format string is printed in place of their corresponding `%d` conversion specifier, as shown.

A declaration consists of a type name followed by a comma-separated list of identifiers having that type. When a declaration is defining one or more objects, it can also contain corresponding initializers. For example:

```
int i = 10, start, j = 20, end;
```

which is equivalent to:

```
int i = 10;
int start;
int j = 20;
int end;
```

> **Style Tip:** By placing each variable in its own separate declaration, with one declaration per source line, you leave room for a trailing comment. You can also cut and paste lines more easily in a full-screen text editor.

Like C++, C99 allows declarations and statements to be intermixed. As a result, the body of `main` in example ba03b above can be written as follows (see directory ba03c):

```
int year = 1992;
printf("The year %d was a leap year.\n", year);

int i = 10;
printf("%d squared = %d\n", i, i * i);
```

however, this feature is not used in this book.

1.4 The Data Types

1.4.1 Arithmetic Types

The set of integer types comprises signed and unsigned versions of `char`, `short int`, `int`, `long int` and `long long int`.[1] In the case of `short int`, `long int`, and `long long int`, the keyword `int` is optional. The set of floating-point types comprises `float`, `double`, and `long double`, all of which are signed.[2]

---

[1] C99 invented the type `long long int`. At least one widely used implementation calls this `__int64` instead.

[2] C99 invented the types `float _Complex`, `double _Complex`, `long double _Complex`, which are optional. See §C.3 for more details.

> **Implementation-Defined Behavior:** The range of values that can be stored in an object of a given arithmetic type.

Standard C specifies a minimum range—and in the case of floating-point types, a minimum precision—that must be supported.

An implementer of a C compiler determines how C's data types are mapped. And provided they meet the minimum range and precision requirements, different implementations (even those running on the same system) may use different mappings. The following table shows mappings commonly used for machines with word sizes as shown:

Table 1-2: Common Arithmetic Type Mappings

Type/Word Size	8-bit	16-bit	32-bit	64-bit
[unsigned] char	8 bits	8 bits	8 bits	8 bits
[unsigned] short [int]	16 bits	16 bits	16 bits	32 bits
[unsigned] [int]	16 bits	16 bits	32 bits	64 bits
[unsigned] long [int]	32 bits	32 bits	32 bits	64 bits
[unsigned] long long [int][C99]	64 bits	64 bits	64 bits	64 bits
float	32 bits	32 bits	32 bits	32 bits
double	64 bits	64 bits	64 bits	64 bits
long double	64 bits	64/80 bits	64/80 bits	64/128 bits

In the case of long double, entries of the form x/y mean "x or y". Remember that these are possible mappings, not required mappings.

Standard C imposes the following minimal requirements on arithmetic types:

- An object of the integer type char must be able to represent every character in the base character set. It must contain at least 8 bits.
- An object of the integer type short must have at least 16 bits. Its size must be greater than or equal to that of a char.
- An object of the integer type int must have at least 16 bits. Its size must be greater than or equal to that of a short.
- An object of the integer type long must have at least 32 bits. Its size must be greater than or equal to that of an int.
- An object of the integer type long long must have at least 64 bits. Its size must be greater than or equal to that of an int.

- An object of the floating type `float` must be able to represent a value with at least six significant digits and with an exponent having a range of at least ±37.
- An object of the floating type `double` must be able to represent a value with at least 10 significant digits and with an exponent having a range of at least ±37. An object of type `double` must be able to store a value with at least the same range and precision as an object of type `float`.
- An object of the floating type `long double` must be able to represent a value with at least 10 significant digits and with an exponent having a range of at least ±37. An object of type `long double` must be able to store a value with at least the same range and precision as an object of type `double`.

A series of predefined *symbolic constants* is provided in the standard library headers `limits.h` and `float.h`.[1] These constants allow us to find out various attributes of each arithmetic type. Obviously, the range and the precision of these types are important to programmers writing code that is to be portable across different machine architectures.

The type name `char` can be misleading. While a `char` variable can, and usually does, contain a character, characters are really represented in computers as small integers. Therefore, we can use a `char` variable to store integers directly. For example:

```
char c = 10;    /* using a char as a small integer */
```

The integer types can be made signed or unsigned by prefixing them with the keyword `signed` or `unsigned`, respectively. Signed and unsigned versions of the same type have identical sizes; however, in the case of `signed`, one bit is interpreted as a sign bit.[2] By default, `short`, `int`, `long`, and `long long` are signed, in which case, using `signed` with them is redundant.

> **Implementation-Defined Behavior:** Whether a *plain* `char` (that is, one without an explicit `signed` or `unsigned` keyword) is signed.

We can manipulate the bits in an integer value via a series of operators; for example, we can shift bits left or right using the binary operators <<, <<=, >>, and >>=, we can mask them using the binary operators &, &=, |, |=, ^, and {^=}, and we can complement them using the unary operator ~. (See §4.6 for more information.)

The following program (see directory ba04) provides some more examples of output formatting:

```
#include <stdio.h>

int main()
{
        int i = 200;
        double d = .125556;
```

[1] Examples of using some of these constants are shown in §6.
[2] Standard C permits both ones- and twos-complement as well as signed-magnitude representation of integers.


```
/*1*/   printf("%d, %x, %X, %o\n", i, i, i, i);
/*2*/   printf("%4d, %04d\n", i, i);
/*3*/   printf("%6.3f\n", d);

        return 0;
}
```

The output produced is:

```
200, c8, C8, 310
 200, 0200
 0.126
```

`printf` can display the values of any integer expression, regardless of its type. We have already learned about `%d`. To display the value of an expression of type `int` or `unsigned int` using octal notation, use `%o`. To get uppercase hex notation, use `%X`; for lowercase hex, use `%x`. To display the value of an expression of type `unsigned int` in decimal, use `%u`. For `long` integers, add an `l` modifier prefix. For example, `%lu` is used to display the value of an expression of type `unsigned long`.

To display the value of an expression of type `char` in its printable form, use `%c`. To see its internal representation, use `%d`, `%o`, `%x`, or `%X`, as appropriate. The value of an expression of type `short` or `unsigned short` is printed using the same conversion specifiers as for `int` or `unsigned int`, respectively, since expressions of type `short` are implicitly converted to `int` across function calls.

A set of conversion specifiers also exists to display the values of any floating-point expression. `%f` is used for expressions of type `double`. To display the value of an expression of type `long double` use `%Lf`.

To display the value of an expression of type `float`, use the same conversion specifiers as for `double`, since expressions of type `float` are implicitly converted to `double` across function calls.

The conversion specifiers `%f` and `%Lf` result in output using fractional notation. To get output containing an exponent, use `%e` or `%Le` instead. The conversion specifiers `%g` and `%Lg` use the shorter of `%f` (or `%e`) and `%Lf` (or `%Le`). To force the 'e' in the exponent to be uppercase, use `E` (or `G`) instead of `e` (or `g`).

The output display field width and precision can be set using a conversion specifier such as `%6.3f`, where the width of the display field will be at least six (more print positions will be used if necessary) and there will be three fractional digits. By default, the conversion specifier `%f` prints six fractional digits.

Exercise 1-8: Find out the size and precision of all the arithmetic types for your implementation. (Hint: look for information in the standard headers `limits.h` and `float.h`.) Is `long double` equivalent to `double`? Are `short`, `int`, and `long` all different, or is there some overlap? Is a plain `char` signed or unsigned? Some compilers provide an option to select the signedness of a plain `char`; does yours?

Exercise 1-9*: Write a program that has three `int` variables named `cost`, `markup`, and `quantity`, and initialize them to $20, $2, and 123, respectively. Print the result of *retail cost* × *quantity*, where *retail cost* equals *cost* + *markup*. (See lab directory lbba01.)

1.4.2 The Boolean Type

C99 added support for a Boolean type. Including the header `stdbool.h` gives us access to the type name `bool` and its two possible values, `true` and `false` (none of which are actually keywords). Note carefully that `true` and

`false` really are synonyms for the integer constants 1 and 0, respectively, so we have only an illusion of a Boolean type, when no such separate type actually exists. Refer to your implementation's documentation for more information on this header.

1.4.3 Enumerated Types

An *enumeration* is made up of a set of named constant values each of which is called an *enumeration constant*. Each distinct enumeration constitutes a different enumerated type. The use of enumerations and their enumeration constants provides a useful amount of abstraction. Let us look at the following (see directory ba06) for some examples:

```
/*1a*/  enum carColor {black, white};
/*1b*/  enum carColor c1 = black;

/*2a*/  enum houseColor {red, green = 12, blue, scarlet = red};
/*2b*/  enum houseColor c2 = green;

/*3*/   c1 = 0;         /* permitted, but illogical */
/*4*/   c1 = white;     /* okay                     */
/*5*/   c1 = blue;      /* permitted, but illogical */
/*6*/   c2 = white;     /* permitted, but illogical */
/*7*/   c2 = blue;      /* okay                     */
/*8*/   c1 = c2;        /* permitted, but illogical */
```

In case 1a, we declare a named enumeration type called `carColor` with two enumeration constants, `black` and `white`. (Enumeration constants are identifiers.) By default, the internal value of an enumeration constant is one more than its predecessor, with the first having value 0.[1] In case 1b, we define `c1` to be a variable of this type, and give it an initial value of `black`. Similarly, in case 2a, we declare an enumeration type, `houseColor`, giving it four enumeration constants; however, `green` is given the value 12, resulting in `blue`'s being 13, while `scarlet` has the same value as `red`. Clearly then, the range of values used by the members in an enumeration can involve gaps and/or duplicates.

Unfortunately, since operations involving enumerations and enumeration constants are *not* strongly checked with respect to type compatibility, situations such as those occurring in cases 3, 5, 6, and 8, are accepted.[2]

> **Tip:** Avoid initializing enumeration constants explicitly or making duplicates unless you have good reason to do so. Assigning a range of values that is non-continuous or that contains duplicates significantly reduces the set of operations that can meaningfully be performed.

[1] This makes them useful as array subscripts, which, as we will learn in §3, begin at zero.
[2] Such strict checking *is* done by a C++ compiler.

Let us consider another example (see directory ba07):

```
/*1*/   enum {male, female};
/*2*/   int sexCode = male;

/*3*/   printf("male   = %d\n", male);
/*4*/   printf("female = %d\n", female);
/*5*/   printf("sexCode = %d\n", sexCode);
```

whose output is as follows:

```
male   = 0
female = 1
sexCode = 0
```

In case 1, we declare an unnamed enumeration type having two enumeration constants; however, later on, we can never define variables of this type. However, as we see in case 2, we can use enumeration constants in any context permitting an integral constant value.

Cases 3 and 4 simply demonstrate that the values of `male` and `female` are 0 and 1, respectively.

In order to share an enumeration type declaration between separately compiled source files, we simply create a user-defined header. Let's call this header Constants.h and put it in a source file of the same name:

```
/* source file Constants.h */

enum SexCode {male, female};
enum CursorDirection {up, down, left, right};
```

To use this header, we simple make it available during compilation much like we do for `stdio.h`, except that we use "…" delimiters rather than <…>, as follows:

```
#include "Constants.h"

void process()
{
        enum SexCode code = male;
        enum CursorDirection dir = left;
        /* … */
}
```

1.4.4 User-Defined Types

We can define an unlimited number of user-defined object types, such as Employee, Publication, Transaction, Message, and Point. We will learn how to do this in §9.

1.5 Literals

Arithmetic literals can have integer or floating-point type, depending on the presence or absence of a decimal point, exponent, and type suffix.

C does not support negative literals. Expressions of the form *-literal* involve the unary minus operator and a non-negative literal.

1.5.1 Integer Literals

An integer literal has a base (or radix), as determined by the presence of a prefix. If an integer literal has a prefix of 0x or 0X, it is interpreted as a hexadecimal (base 16) number and is permitted to contain the letters a–f and A–F, as well as the digits 0–9. If the prefix is just 0 (zero), the number is interpreted as octal (base 8) and only the digits 0–7 are permitted. All other integer literals are deemed to be decimal (base 10) and can contain only the digits 0–9. There is no support for expressing numbers in binary (base 2).

Since a literal is an expression, and each expression has a type, a literal has a type as well as a value.

> **Implementation-Defined Behavior:** The type of a given integer literal, since it depends on the mapping of the integer types by the compiler.

The following table shows the steps used by a compiler to find the type of an integer literal:

Table 1-3: Integer Literal Typing Steps

Type/Base	Decimal	Octal/Hex
`int`	1	1
`unsigned int`	—	2
`long int`	2	3
`unsigned long int`	—	4
`long long int`	3	5
`unsigned long long int`	—	6

If the value of a decimal literal can be represented as an `int`, that is its type. If it cannot, and it fits in a `long int`, that is its type. If it cannot, and it fits in a `long long int`, that is its type.

> **Undefined Behavior:** If the value of a decimal integer literal cannot be represented as `long long int`.

For octal and hexadecimal literals, extra steps involving `unsigned` types are added.[1]

[1] This means that a decimal literal with a given value can have a different type than the same-valued literal expressed in octal or hex!

An integer literal can explicitly be given the type `long int` by the addition of a suffix of L or `l`. An integer literal can explicitly be given the type `long long int` by the addition of a suffix of LL or `ll`. A U or u suffix makes an integer literal `unsigned`. This can be combined with `l`, L, `ll`, or LL.

> **Style Tip:** When explicitly typing integer literals with a `long` or `long long` suffix, use L instead of `l` since the latter can easily be mistaken for the digit 1.

The following are examples of integer literals:

```
123456          /* decimal, int or possibly long int */
01234           /* octal, int                        */
0x09aB          /* hex, int                          */
0XfFc5          /* hex, int or possibly unsigned int */
5L              /* decimal, long int                 */
0xAb2L          /* hex, long int                     */
065U            /* octal, unsigned int               */
0x234uL         /* hex, unsigned long int            */
1234567890LL    /* decimal, long long int            */
```

1.5.2 Character Literals

Since characters really are represented as integers, variables of type `char` can be initialized with integer expressions. However, initializing a `char` variable with the value 65 on a system using the ASCII character set requires that the reader know that the internal value of A is 65. It also makes the code ASCII-specific. To allow for readability and portability, we can write such values in the form of *character literals* using the form `'`*x*`'`. A character literal has type `int`[1] and its value is that of *x* in the machine's character set.

The following are examples of character literals:

```
'A'  '+'  'ß'  'ñ'  'æ'  '\n'  '\''  '\xff'
```

As we can see, a character literal can contain any of the escape sequences or a printable character. The single quote sequence is needed only when writing a character literal containing a single quote.

1.5.3 Floating-Point Literals

A floating-point literal must contain a decimal point, an exponent, or both. The exponent can be written using either an upper- or lowercase E, and it can contain a leading sign. Both the value and exponent parts are interpreted as decimal. (C99 added support for floating-point literals written in hexadecimal and having an exponent written using an upper- or lowercase P. Refer to your compiler manual for more information.)

An unsuffixed floating-point literal has type `double`. A suffix of F or `f` indicates type `float`, while a suffix of L or `l` indicates type `long double`. Unlike integer literals, the compiler does not use a table of steps to determine the type of a floating-point literal; the suffix, or absence thereof, says it all.

[1] In C++, a character literal has type `char`.

> **Style Tip:** When typing a `long double` literal, use `L` instead of `l` since the latter can easily be mistaken for the digit 1.

The following are examples of floating-point literals:

```
.952            /* double      */
3e34            /* double      */
1.23E5          /* double      */
345.F           /* float       */
3.456e+4f       /* float       */
321e-6L         /* long double */
435.54l         /* long double */
```

> **Exercise 1-10*:** Study the following program and predict what will happen when it is run; then run it. Can you explain the output? Look carefully at what was promised to `printf`, and compare that to what was actually passed (see labs directory lbba08).
>
> ```
> #include <stdio.h>
>
> int main()
> {
> /*1*/   printf("%d\n", 12345, 321);
> /*2*/   printf("%d %d\n", 654);
> /*3*/   printf("%d %d\n", 1.234, 23);
> /*4*/   printf("%d\n", 1.6 * 5);
>         return 0;
> }
> ```

1.5.4 Boolean Literals

See §1.4.2.

1.5.5 String Literals

As we learned in §1.3, a string literal is a sequence of characters surrounded by double quotes, as in `"hello"`. The string literal `""` represents the empty string. A string literal has type "array of *n* `char`", where *n* is one more than the number of characters shown. (For example, `"hello"` is an unnamed array of six characters.[1])

Adjacent string literals are concatenated by the compiler.

> **Unspecified Behavior:** Whether two string literals containing the same characters are stored in distinct arrays.

[1] We'll learn more about arrays of characters and how string literals are stored, in §3.3.

Although programmers tend to think of string literals as being constants, some implementations store them in read/write memory.

> **Undefined Behavior:** Whether a string literal can be modified.

1.5.6 Compound Literals

C99 introduced the idea of a *compound literal*, which, as its name suggests, is intended to allow unnamed literals for objects containing multiple items, such as arrays and structures. The general form of a compound literal is:

```
( type-name ){ initializer-list }
```

where the initializer list can optional be followed by a comma. For examples of compound literals of array type, see §3.2; of structure type, see §9.3; and of union type, see §9.11. Although a scalar literal, such as `12345`, can be written using a compound literal, as in `(int){12345}`, this is far less easy to read than the simple scalar literal.

1.6 Type Qualifiers

C has a number of type qualifiers:[1] `const`, `restrict`, and `volatile`. They can be used together. As the name implies, a type qualifier somehow qualifies (by restricting or controlling) the way in which an object of that type can be accessed. These qualifiers can be used together.

1.6.1 The `const` Qualifier

When the `const` qualifier is applied to an object, it prohibits that object from being modified. For example:

```
const int maximum = 100;
const double pi = 3.1415926;

maximum = 200;  /* error */
```

While a `const`-qualified object cannot be modified, it can be (indeed, it must be) initialized via an initializer, as shown in the definition of `maximum` and `pi` above.

It is important to understand that `const` does not guarantee that an object will be stored in read-only memory at run time. It might or might not be, at the implementation's pleasure.

The obvious application of `const`-qualified "variables" is in defining symbolic constants, those sensible names given to obscure physical constants. In order to share a set of symbolic constants, we simply create a user-defined header, just as we did in §1.4.3. Then if the need arises to change the initial value of a `const`-qualified object, we can simply change its initializer in the header and recompile all source files that include it.

As we shall see throughout this book, `const` can be used very effectively in the context of data pointers (§7.5.1).

1.6.2 The `volatile` Qualifier

The `volatile` qualifier tells the compiler that the object so qualified might be accessed by a hardware device or by multiple threads of execution at the same time in an asynchronous manner. For example, the object might be the receive buffer of a serial port. Alternatively, the object might reside in shared memory and be used as a synchronizing indicator between cooperating programs.

---

[1] C89 invented `volatile` and adapted `const` from C++. C99 invented the type qualifier `restrict`.

Essentially, `volatile` is a directive to the compiler to disable certain optimizations in such a way that *every* time there is a logical access to a `volatile`-qualified object, the object is physically accessed as well.

Consider the following example (see directory ba09) that computes $y = 2x^2 + 5x + 6$ in two different ways:

```
volatile int x;

void f1()
{
        int y = (2 * x * x) + (5 * x) + 6;
}

void f2()
{
        int t = x;      /* snapshot copy of x */
        int y = (2 * t * t) + (5 * t) + 6;
}
```

The result computed in `f1` is unreliable, because the `volatile` qualifier on `x` requires the compiler to fetch `x`'s value three times, yet its value may have changed between fetches. By accessing `x` once and storing a copy of it in the non-`volatile` object `t`, `f2` produces a value consistent for any given value of `x`.[1]

In this example, `x` is defined outside of any function; we'll learn what this means in §5.8.

1.6.3 The `restrict` Qualifier

As this qualifier can only be used in the context of data pointers, discussion of it is deferred until §7.5.2.

1.7 Type Synonyms

The `typedef` keyword provides the ability to create a synonym for another type. For example:

```
typedef unsigned int Counter;
```

Here, we have "invented" the type `Counter` by making it a synonym for the type `unsigned int`. To create a synonym for a type, write a declaration for an identifier of that type and prepend the keyword `typedef`; the identifier now becomes the new type name.

Once created, a type name can be used in any context in which the underlying type may occur—for example:

```
Counter total = 0;
```

Why invent a type synonym if we can always write the underlying "real" type? `typedef` allows us to invent an abstract type name. If we give it a descriptive name and promise certain properties, we can use that type very effectively without knowing exactly its underlying type. For example, we are writing code that is to be ported across a number of platforms and we need various counters. The value of a counter is always in the range 0–1000. Of course, counters are whole numbers. By inventing the type name `Counter`, we allow the underlying type to be changed when the code is ported. We might wish to map the type to a signed or unsigned `short`, `int`, or `long` for size or speed reasons, depending on the target machine's architecture.

---

[1] This approach works only if `x` can be accessed in an atomic operation.

There are also occasions where we receive a value from some library function, store it, and later pass it back to another library function without ever using it directly. In such cases, we don't need to know what type that value has. By hiding the underlying type information, we allow the author of the library to change the actual type as long as he maps the type synonym to the new type. Type qualifiers can be included in the type underlying a typedef; for example:

```
typedef const unsigned int Constant;

Constant i = 10;
```

Since type synonyms are intended to be shared between separately compiled source files, their definitions should be placed inside a user-defined header.

Tip: If you have to know the underlying type of a type synonym in order to use objects of that type effectively, the type synonym is useless.

Exercise 1-11: Standard C requires numerous type synonyms to be defined in one or more standard headers. Read about the following abstract data types in your library documentation:

Table 1-4: Examples of Abstract Types Defined in Standard Headers

Type	Header
`clock_t`	`time.h`
`FILE`	`stdio.h`
`fpos_t`	`stdio.h`
`size_t`	`stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, among others
`wchar_t`	`stddef.h`, `stdlib.h`, `wchar.h`

1.8 Automatic Variables

All of the variables used so far have been defined inside the braces delimiting the body of a function. Variables defined within a function are local to that function and are not accessible by name from within other functions. Ordinarily, they are created each time their parent function is invoked, and they disappear when that function terminates. They are known as *automatic variables* because they automatically come and go.[1]

Typically, the cost of allocating memory for automatic variables is independent of the amount being allocated; that is, it costs the same at run time to allocate one variable as it does to allocate 100 or 1,000.

[1] In 5.3, we'll see how to define variables that live for the whole life of a program.

Undefined Behavior: By default, the initial value of an automatic variable is undefined; whatever garbage happens to be lying around in the memory allocated becomes the initial value of the newly allocated variable. Beware!

Tip: A common source of error is the failure to initialize an automatic variable.

Consider the following program (see directory ba11):

```
#include <stdio.h>

int main()
{
/*1*/   signed int si = 100;    /* signed is redundant */
/*2*/   unsigned int ui = -20;  /* negative to unsigned */
/*3*/   int i = 1000000;        /* will this number fit? */
/*4*/   unsigned long ul;       /* undefined initial value */
/*5*/   double d = si + ui;     /* mixed-mode arithmetic */

/* By now, memory has been allocated for all the variables */

/*6*/   printf("ui = %u\n", ui);
/*7*/   printf("i = %d\n", i);
/*8*/   printf("si + ui = %u\n", si + ui);
/*9*/   printf("d = %f\n", d);

        return 0;
} /* automatic variables are destroyed here */
```

The output produced on one system (that used 16-bit, twos-complement arithmetic) was:

```
ui = 65516
i = 16960
si + ui = 80
d = 80.000000
```

When run on another system (that used 32-bit, twos-complement arithmetic), the output was:

```
ui = 4294967276
i = 1000000
si + ui = 80
d = 80.000000
```

In cases 1 and 2, the variables are explicitly initialized. Of course, since they are created at run time, on entry to this function, they also are initialized at run time.

In case 2, an unsigned integer is being initialized with a negative value. While this is permitted, the value actually stored is some large unsigned number, as shown by the output produced in case 6.

> **Tip:** Be careful when mixing unsigned integers with signed integers that have negative values. While the result is predicable, it may be surprising.

In case 3, we are attempting to store a large number in a signed `int`. And since an object of this type is not required to have more than 16 bits, truncation can occur, as shown in the output produced by case 7 on the 16-bit system. Whether a compiler warns about this possibility is a *quality of implementation* issue. That is, it is a marketplace issue, not something mandated by the C standard.

In case 4, the keyword `int` is implied and the initial value of `ul` is undefined. In case 5, the initializer consists of an expression involving a computation. Such an expression can involve any terms whose value can be computed at run time. The only criterion is that any variables referenced in that expression must already have been defined and initialized.

The initializer in case 5 is worth further mention. Because this expression involves terms having different (but compatible) arithmetic types, type promotion is involved; the value of the `signed int` is promoted to `unsigned int` and the type of the result of the addition is `unsigned int`. However, this type cannot accurately store the result, resulting in "modulo wrap around".[1] The resulting value, 80, is then used to initialize the `double` variable `d`. The important aspect here is that even though `d` can represent the correct result of adding `si` and `ui`, the sum is truncated before being used to initialize `d`.

In the four cases involving explicit initialization, we could have achieved the same result by omitting the initializers and using equivalent assignment statements instead; it's a matter of style as to which you use.

It is important to note that the order in which we define variables has no effect on where they are stored in memory relative to one another. Their relative locations are determined by the compiler and are unspecified by Standard C.

1.9 Operator Precedence

When an expression involves more than one operator, the compiler determines the order in which it groups terms, based on the *precedence table*.[2] Consider the following example (see directory ba12):

```
#include <stdio.h>

int main()
{
        int i = 5;
        long int l = 500;
        float f = 3.58345F;
        double d;
```

[1] It is impossible to get overflow with unsigned integer arithmetic.

[2] This table is shown in Annex A, along with a discussion of operator precedence and associativity.


```
/*1*/   d = i + l * f;              /* d = i + (l * f) */
        printf("d = %.1f\n", d);

/*2*/   d = (i + l) * f;
        printf("d = %.*f\n", 2, d);

/*3*/   d = i + l + f;              /* d = (i + l) + f */
        printf("d = %*.1f\n", 7, d);

/*4*/   ((d) = (((i) + (l)) + (f)));
        printf("d = %*.*f\n", 8, 3, d);

        return 0;
}
```

The output produced is:

```
d = 1796.7
d = 1809.64
d =   508.6
d =  508.583
```

In case 1, multiplication has higher precedence than addition, resulting in the value of `l` being multiplied by the value of `f`, and the result being added to the value of `i`. `d` is displayed with one decimal place, as indicated by the `.1` precision.

The default precedence and associativity can be overridden via the use of grouping parentheses, as shown in case 2. Here, the values of `i` and `l` are added, and the result is multiplied by the value of `f`. `d` is displayed with the number of decimal places being determined at run time, as indicated by the `.*` precision. In this situation, the next argument is expected to have type `int`, and it is interpreted as the precision.

In case 3, the two addition operators have the same precedence, since they are the same operator. However, these operators have left to right *associativity*. That is, the left one is given higher precedence. Therefore, the values of `i` and `l` are added, and the result is then added to the value of `f`. `d` is displayed with one decimal place and right justified in a column whose width is determined at run time, as indicated by the width *. In this situation, the next argument is expected to have type `int`, and it is interpreted as the width.

In case 4, all the grouping parentheses are redundant.

> **Style Tip:** While the use of redundant grouping parentheses can sometimes help readability, excessive use can hinder it.

In some languages, the order in which terms of an expression are evaluated is tied to the grouping of terms as specified by operator precedence. This is *not* the case in C.

> **Unspecified Behavior:** The order in which most individual terms in an expression are evaluated.

Consider the following example:

```
x = f() + g() * h();
```

The precedence is quite clear; multiplication wins out over addition, as demonstrated by the corresponding parse tree:

```
    +
   / \
  /   \
f()    *
      / \
     /   \
   g()   h()
```

However, in what order are the three terms themselves evaluated? That is, in what order are the three functions called? If the three functions behave differently when called in one order versus another, the order of their evaluation is important. The only way to make the order of evaluation predictable is to break the statement into multiple statements.[1] For example, the steps:

```
x = g();
x = x * h();
x = x + f();
```

result in the same precedence as before, but with the functions called in the guaranteed order `g()`, `h()`, and `f()`.

> **Tip:** A common error is to rely on the order of evaluation of terms across a given operator when no such guarantee is given by the language.

1.10 Type Conversion

Like most languages, C permits expressions to contain terms of different types, provided the types are compatible. Consider the following mixed-mode expression in which `d` has type `double`, `i` has type `int`, `l` has type `long int`, and `f` has type `float`:

```
d = i + l + f;      /* d = (i + l) + f */
```

The precedence table determines how pairs of terms are grouped. As a result of that grouping, the type of each subexpression is determined. In the example above, since `i` and `l` have different (but compatible) types, implicit conversion of the value of `i` to `long int` occurs, and the result of the addition has type `long int`. Then that value is converted to type `float` so it can be added to `f`, producing a result of type `float`. The value of this result is then converted to type `double`, so it can be assigned to `d`.

[1] For information on when the compiler is required to perform certain operations, see "Sequence Points" in the Advanced-C book.

The order in which we write the terms in this assignment expression can be important. For example, the following are subtly different:

```
/*1*/   d = i + l + f;          /* d = (i + l) + f */
/*2*/   d = f + i + l;          /* d = (f + i) + l */
```

In case 2, the value of `i` is converted to type `float` and then added to `f`, producing a result of type `float`. Then, the value of `l` is converted to type `float` and added to the result of the previous addition. For many values of `i`, `l`, and `f`, the two versions will produce the same result. However, consider case 1 when `l` contains the largest value that can be stored in an object of type `long int`, and the value of `i` is 1 or more. When they are added, overflow occurs, since the result cannot be represented in the type `long int`. If, however, the expression is rearranged as in case 2, all intermediate results are performed in type `float`, so no integer overflow can occur.[1]

Controlling the type of intermediate results by rearranging the expressions is poor style. Instead, it is better to use explicit type conversion. This is done via the *cast* operator, which has the following form:

(*type-name*) *operand*

In the case of `(int)x`, an unnamed object of type `int` is created, and its value is that of `x` converted to type `int`. The type and value of `x` are unchanged. We can use a cast in our previous example to avoid the possibility of integer overflow, as follows:

```
/*1*/   d = (float)i + l + f;
/*2*/   d = i + (float)l + f;
/*3*/   d = (float)i + (float)l + f;
```

In case 1, the value of `i` is explicitly converted to type `float`, resulting in the value of `l`'s being implicitly converted to type `float` as well. In case 2, the value of `l` is explicitly converted to type `float`, resulting in the value of `i`'s also being implicitly converted to type `float`. In case 3, both `i` and `l` are explicitly cast. All three cases are equivalent.

The value of an expression having any arithmetic type can be cast explicitly to any arithmetic type; however, it may result in a loss of precision, for example, when a `double` is cast to a `float` or an `int`. An expression can be cast to its own type, although such a cast has no effect.

Style Tip: Avoid unnecessary casts because they detract from readability.

[1] In mathematics, addition is both associative and commutative, and overflow cannot occur. However, when numbers are represented in a machine, a finite range is used, so overflow becomes a possibility.

3. Arrays and Strings

In this chapter, we will learn how to define and use arrays. While only a few types are used in the examples, the principles can be applied to arrays of objects of any data type, including user-defined types. We'll also look at string manipulation, and some character testing functions from the standard library.

3.1 Introduction

There are four main differences between using arrays in C and in many other languages.

1. In array definitions and subscripts, square brackets are used instead of parentheses. This is a minor issue, and a compiler will produce an error if we inadvertently use parentheses instead.
2. Each dimension in a multidimensional array is written with its own set of brackets. This allows an array to be subscripted with less than the maximum number of dimensions, something not possible in many other languages.
3. Array elements begin at subscript 0, not 1. This may or may not be a major problem, depending on one's language background. It can result in "off by one" errors. For example, if we access an array of 10 elements by subscripting from 1 to 10, we miss element 0 and we access the nonexistent element 10.

> **Undefined Behavior:** Attempting to access an out-of-bounds array element.

   And to make it really interesting, for the most part, C compilers cannot easily enforce array bounds checking, so they don't.
4. Subscripting involves the use of `[]`. In this context, `[]` represents an operator, so it is subject to the rules of precedence and associativity.

> **Tip:** Programmers new to C often forget that for an array with a dimension of size *n*, the range of valid subscripts for that dimension is 0 through *n*-1.
>
> **Tip:** The following expressions exhibit undefined behavior because the order of evaluation across the assignment operator is undefined: `a[i] = b[i++]` and `a[i++] = b[i]`. Similarly, `x[i][i++]` and `x[++i][i]` exhibit undefined behavior because the order of evaluation across the subscript operator is undefined.

Some array definitions and subscript expressions follow (see directory ar01):

```
int main()
{
        float f[10];            /* [0] … [9]       */
        long l[3][5];           /* [0][0] … [2][4] */
        int i, j, k;

        for (k = 0; k < 10; ++k)
        {
                f[k] = 0.0F;
        }

        for (i = 0; i <= 2; ++i)
        {
                for (j = 0; j <= 4; ++j)
                {
                        l[i][j] = 10;
                }
        }

        return 0;
}
```

The number of elements in each dimension of an array declaration must be a compile-time integer constant expression with a value greater than zero. In particular, note that, unlike most other languages, the dimension expression can contain any arithmetic operators that make sense with integer constants. For example, `f` could have been declared using `float f[5 + 5]`.[1]

Prior to C99, variable-length arrays were not permitted by the language; instead, they were (and still can) be implemented at run time by using various library functions.[2] For a discussion of C99's addition, see §3.8.

In C, arrays are stored in row-major order. That is, elements are stored in memory such that the right-most subscript varies the quickest, like BASIC, COBOL, and Pascal, but unlike FORTRAN.

C places no limit on the number of dimensions; we are limited only by the amount of memory available.

If an array name is defined with the `const` qualifier, all of its elements are `const`-qualified.

Note that in the example above, the braces in the single and nested `for`s are all optional, since the body of each is only a single statement.

> **Exercise 3-1*:** Define an `int` array with only one element. Initialize it to the value 100, and display its value using `printf`. (See labs directory lbar03.)
>
> **Exercise 3-2*:** Define an `int` array of 10 elements. Display the value of elements 15, 100, and -5, all of which are nonexistent. Do you get any compilation or run-time errors? If so, can you explain them? (See labs directory lbar04.)

[1] The importance of this capability will be seen in §6.2.
[2] This capability is discussed in detail in §7.11.

3.2 Initialization

An array is an *aggregate*; that is, an object containing subobjects, each of which has its own value. While we can set the value of each element individually via assignment, that approach is tedious. Instead, we can use an *initializer*, which consists of a brace-delimited list of expressions; for example (see directory ar05):

```
int counts[5] = {10, 35, 56, 76, 12};
double value[3] = {1.2, 2.3, 3.4};

enum Color {red, brown, green, blue};
enum Color hue[5] = {green, blue, red, brown, red};
```

When we are initializing a multidimensional array, each dimension has its own initializer list. For example:

```
int iarray1[2][3] = {
        {1, 2, 3},
        {5, 4, 3}
};
```

The array definition is read as "`iarray1` is an array containing two elements, each of which is an array containing three elements". Therefore, the large initializer list contains two subinitializer lists, each of which corresponds to a row. A 3-D array is initialized in a similar manner:

```
int iarray2[2][3][2] = {
        {{1, 2}, {4, 5}, {2, 3}},
        {{6, 1}, {9, 3}, {0, 2}}
};
```

The array definition is read as "`iarray2` is an array containing two elements, each of which is an array containing three elements, each of which is an array containing two elements".

Some languages, notably FORTRAN, provide a repetition count when identical consecutive initializing expressions are required. C does not provide such a capability. If we wish to initialize each element of a 100-element array with the value 1, for example, we must specify an initializer list with 100 expressions containing that value.

An array initializer list need not contain values for every element. For example:

```
int total[4] = {123, 62};       /* 2 trailing 0s   */

float f[3][3] = {
        {1.2F, 3.4F},           /* 1 trailing 0.0  */
        {9.8F}                  /* 2 trailing 0.0s */
                                /* 3 trailing 0.0s */
};

int table[5];                   /* all undefined */
```

Any trailing elements not explicitly initialized take on the value of zero cast to their type. However, this applies only to partial initializer lists. If the initializer list is completely empty, as is the case with `table`, the initial value of each element is undefined.[1]

Only trailing element values can be omitted. That is, initializer lists like the following are prohibited:

```
int total[4] = {, 62, , 4};     /* error */
```

In an array definition, we can omit the size of the first dimension provided an initializer list is present. For example (see directory ar06):

```
/*1*/   int value[] = {10, 20, 30, 40};

/*2*/   long table[][3] = {
                {1, 2},         /* 1 trailing 0 */
                {4}             /* 2 trailing 0s */
        };
```

In case 1, the compiler determines from the initializer that the array has four elements. Similarly, in case 2, it determines that `table` has two rows, since it has two subinitializer lists. However, since the second dimension promises three columns per row, the compiler deduces that trailing initializers have been omitted, resulting in default values of zero.

C99 enhanced initialization with the addition of *designated initializers*. Here is an example of their use with arrays (see directory ar12):

```
/*1*/   int counts[5] = {[2] = 5, [4] = 2, [0] = 4};
/*2*/   int vals[] = {[3] = 7, 9};
```

In case 1, the number of elements is set to five, with three of those elements designated using explicit initializers. The unspecifed elements take on the value zero, resulting in an array containing 4, 0, 5, 0, 2.

In case 2, the number of elements is not specified. The highest element designated explicitly is three, but the initializer following that is implicitly for element 4, so the compiler deduces that the array has five elements, 0–4. The first three (unspecifed) elements take on the value zero, resulting in an array containing 0, 0, 0, 7, 9.

C99 also enhanced initialization with the addition of compound initializers (see §1.5.6). A compound initializer can contain designated initializers. Here is an example of their use with arrays (see directory ba13):

```
char c1 = 'C'; char c2 = 'B';
const char *p = (const char []){'A', [2] = c1, '\0', [1] = c2};

int *globalP = (int []){9, 2};   // initializer must be constant
```

3.3 Strings

Unlike some languages, C has no built-in string type, per se. Instead, in C, a string is an array of characters terminated by a *null character*, a character with an internal value of zero.[1] A string literal also has this form.

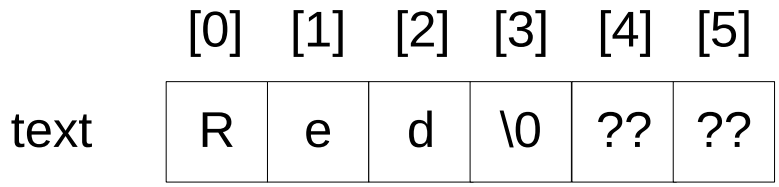
[1] As we'll learn in §4.7, this is only true for automatic variables.

Consider the following simple example (see directory ar02):

```
#include <stdio.h>

int main()
{
/*1*/   char text[6];

        text[1] = 'e';
        text[2] = 'd';
        text[0] = 'R';
/*2*/   text[3] = '\0';
```




```

/*3*/   printf("text contains %s\n", text);
        return 0;
}
```

The output produced is:

```
text contains >Red<
```

In case 1, `text` is declared with size 6, which allows for a 5-character name plus a null character terminator. We then proceed to initialize elements of that array in arbitrary order and, finally, in case 2, we store a null character.[2] Because the array is an automatic object, the initial value of its elements is undefined. If fewer than six characters are stored into the array, those elements not overwritten continue to contain their undefined initial values. However, provided we never try to process characters beyond the null terminator, their residual values are irrelevant.

In case 3, we display the contents of the string stored in the array. As we can see, using the `%s` conversion specifier, `printf` displays each character until it comes across a null terminator, in which case, it stops.

A common problem with string handling is forgetting to append the null terminator, in which case, operations like that in case 3 above keep right on going, walking off the end of the array into the memory that follows. Eventually, they might find something that looks like a null terminator, but by then they will have processed too many characters. If they don't find a terminator, a fatal run-time error can occur when they try to march outside of the memory allocated to their parent program.

---

[1] Programmers are free to construct strings in other ways, such as by using fixed-length arrays of characters with trailing space padding. However, if they do so, they will have to provide their own family of functions to process such strings, since the string functions in the standard library expect the null character terminator to be present.

[2] In §1.5.2, we learned about character constants and how a character value can be expressed using an octal or hex escape sequence. `'\0'` represents a character with octal value 0, which, of course, is also decimal 0.

Tip: When creating or copying a string, don't forget to terminate it with a null character.

When initializing a character array with a string, we must remember to append a null character. For example:

```
char name[5] = {'J', 'a', 'n', 'e', '\0'};

char names[3][5] = {
        {'M', 'a', 'r', 'y', '\0'},
        {'J', 'o', 'h', 'n', '\0'},
        {'J', 'a', 'c', 'k', '\0'}
};
```

However, such initializers are tedious to write and to read; fortunately, they can be written in the following abbreviated manner:

```
char name[5] = "Jane";

char names[3][5] = {
        "Mary",
        "John",
        "Jack"
};
```

In this example, each list is replaced with a string literal containing the same characters, except that the trailing null is implied rather than being written explicitly. The delimiting braces around each string literal are optional and usually are omitted, resulting in a much simpler construct.

As we learned earlier, we can omit the size of the first dimension provided an initializer list is present. For example (see directory ar06):

```
/*3*/   char name[] = "Jackson";

/*4*/   char season[][7] = {
                "Summer",
                "Autumn",
                "Winter",
                "Spring"
        };
```

Exercise 3-3*: Define a `char` array of 10 elements, and initialize it with five characters. Do not append a trailing `'\0'` to the string. Display the array using `printf`, and explain the output. (See labs directory lbar05.)

Exercise 3-4*: How much storage space is taken up by the following strings: `"Europe"`, `""`, `"\0"`, and `"\0abc\0"`? (See labs directory lbar06.)

Exercise 3-5*: Define a 4×30 `char` array, and, using an initializer list, initialize it with the names of four people. Loop through the array displaying each row number and name as you go. (See labs directory lbar01.)

3.4 Array Manipulation

Unlike many languages, C does not permit arrays to be manipulated as a whole. Earlier in this chapter, we passed a string to `printf`. However, what really happened was that the compiler took the address of the first element of the array and passed that to `printf`. No information about the number of dimensions or the size of each dimension was made available. We explain this by saying that the name of the array was converted to the address of the first element.

Consider the following example in which we try to deal with arrays as a whole:

```
        int i1[5], i2[5];

/*1*/   i1 = i2;        /* compilation error */

/*2*/   if (i1 == i2)   /* no error, but … */
        {
                /* … */
        }
```

The assignment in case 1 is rejected even though it seems the obvious way to copy one array to another. The reason is that both array names are converted to the addresses of their first elements. And since this results in the left operand's being a constant—the address of any object is constant for the life of that object—the assignment is rejected.

Unfortunately, case 2 is accepted by the compiler, but does not produce the (presumably) intended result. Instead, the addresses of the first elements of the two arrays are being compared. And since the arrays are distinct objects, their first elements must have different addresses, and thus the comparison always tests false.[1]

> **Tip:** The C language provides no way to manipulate arrays as a whole!

How then can we deal with arrays as a whole? The answer is: by writing a function that manipulates an array an element at a time, or by calling a library function that does that for us. For example (see directory ar07):

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

int main()
{
        int i1[5] = {10, 20, 30, 40, 50}, i2[5];
        int i;

/*1*/   memcpy(i2, i1, 20);             /* copy 20 bytes from i1 to i2 */
```

[1] We'll revisit this issue in §7.4.

```
        for (i = 0; i < 5; ++i)
        {
                printf(" %d", i2[i]);
        }
        printf("\n");

/*2*/   if (memcmp(i1, i2, 20) == 0)    /* compare the first 20 bytes */
        {
                printf("arrays compare equal\n");
        }
        else
        {
                printf("arrays compare unequal\n");
        }

        return 0;
}
```

The header `string.h` provides access to a family of memory-manipulation functions whose names begin with `mem`. `memcpy` copies a given number of bytes from a given source location (the second argument) to a given destination location (the first argument). Of course, the size of the array in bytes can vary across implementations—this example assumes that an `int` is 4 bytes. (A more abstract—as well as portable—technique for calculating such sizes is shown in §3.6.) The `mem*` functions traffic in raw memory; they can handle any types.[1]

Table 3-1: Memory Manipulation Routines in string.h

Name	Purpose
`memchr`	Locate a character in a block of memory.
`memcmp`	Compare two blocks of memory.
`memcpy`	Copy a block of memory.
`memmove`	Copy a block of memory and handle overlap.
`memset`	Initialize a block of memory.

For more details on these functions, refer to your compiler's library manual.

It is important to note that the convention of using null terminators applies only to arrays of `char`. When processing arrays of other types, we must know the number of elements. In fact, `printf` has no way to display the contents of an array except for an array of `char`, and then only if a trailing null is present. To display the contents of an array of `double`, for example, we must use a loop and display the value of each element individually. For example (see directory ar03):

[1] As to how they do this is discussed in §7.12.


```
#include <stdio.h>

int main()
{
        double values[10];

        /* initialize array here somehow */

        int i;
        for (i = 0; i <= 9; ++i)
        {
                printf("values[%d] = %f\n", i, values[i]);
        }

        return 0;
}
```

This long-winded approach is necessary since I/O is not part of the C language; there simply is no way for `printf` to figure out the size of an array being passed to it.

3.5 String Manipulation

The following examle demonstrates the use of some string processing functions (see directory ar08a):

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

int main()
{
        char name1[21], name2[21];
        int length;
        int i;

/*1*/   printf("Please enter a name (20 chars max): ");
/*2*/   scanf("%20s", name1);
/*3*/   length = strlen(name1);
        printf("The name %s has length %d\n", name1, length);
```

```
/*4*/   if ((i = strcmp(name1, "John")) == 0)
        {
                printf("I see your name is John\n");
        }
        else if (i < 0)
        {
                printf("%s < John, alphabetically\n", name1);
        }
        else    /* i > 0 */
        {
                printf("%s > John, alphabetically\n", name1);
        }

/*5*/   strcpy(name2, name1);   /* copies name1 to name2 */
        printf("name2 contains %s\n", name2);

        return 0;
}
```

Some input and the corresponding output are:

```
Please enter a name (20 chars max): James
The name James has length 5
James < John, alphabetically
name2 contains James

Please enter a name (20 chars max): Mary Smith
The name Mary has length 4
Mary > John, alphabetically
name2 contains Mary
```

In case 2, we call `scanf` to read in a string using the conversion specifier `%s`, and automatically add a null terminator to it. By restricting the input to 20 characters maximum, we allow room for a trailing null. In case 3, we call `strlen` to compute the length of the string stored in the array `name1`.

Note the second lot of output. `scanf` terminates its scan when either 20 characters have been entered, or a white-space character has been seen. In the case of the input "Mary Smith", the text "Mary" is read in and stored by `scanf`; however, the remaining characters are left in the input buffer.

> **Tip:** The length returned by `strlen` does not include the null terminator.

In case 4, `strcmp` is used to compare two strings. If the first string has the exact same contents as the second string, this function returns zero. If the first string is lexically less than the second, the function returns a negative value. Otherwise, it returns a positive value. The strings are compared on a character-by-character basis. Two strings must have exactly the same length before they can be equal. The comparison is case-sensitive.

> **Style Tip:** When testing for string equality, it is far more readable and far less error-prone to use something like `strcmp(s1,s2) == 0` rather than `!strcmp(s1,s2)`.

In case 5, `strcpy` is used to copy a complete string, including its trailing null. It is the programmer's responsibility to make sure the destination array is large enough to accommodate the source array and its terminating null.

> **Tip:** `strcpy` copies from its second argument to its first. Calling it with the arguments inadvertently reversed will go undetected by the compiler and may result in undefined behavior.
>
> **Tip:** A common source of error with `strcpy` is to forget to allow room in the destination array for the trailing null.

Table 3-2: String Manipulation Routines in string.h

Name	Purpose
`strcpy`	Copy one string to another.
`strncpy`	Copy the leading part of one string to another.
`strcat`	Concatenate two strings.
`strncat`	Concatenate the leading part of one string to another.
`strcmp`	Compare two strings.
`strncmp`	Compare the leading parts of two strings.
`strcoll`	Compare two strings in a locale-specific manner.
`strchr`	Locate the first occurrence of a character in a string.
`strrchr`	Locate the last occurrence of a character in a string.
`strstr`	Search a string for a substring.
`strcspn`	Match a string from a set of characters.
`strpbrk`	Search a string using a set of characters.
`strspn`	Match a string from a set of characters.
`strtok`	Break a string into specified tokens.

Name	Purpose
strlen	Find the length of a string.
strxfrm	Transform a string.

For more details on these functions, refer to your compiler's library manual.

A number of functions are also provided that convert strings of digits to numbers. For example (see directory ar09):

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

int main()
{
        char number[21];

        printf("Enter a number: ");
        scanf("%20s", number);

/*1*/   printf("The number is %d\n", atoi(number));
/*2*/   printf("The number is %f\n", strtod(number, 0));

        return 0;
}
```

An example of some input and the corresponding output is:

```
Enter a number: 1.23456
The number is 1
The number is 1.234560
```

In case 1, `atoi` converts a string to an `int`. In case 2, `strtod` converts a string to a `double`. (The second argument to `strtod` can be other than zero. However, until we learn more about addresses, possible alternate values are of no interest.)

Table 3-3: String Conversion Routines in `stdlib.h`

Name	Purpose
atof	Convert a number string to `double`.
atoi	Convert a number string to `int`.
atol	Convert a number string to `long int`.
atoll[C99]	Convert a number string to `long long int`.

Name	Purpose
`strtod`	Convert a number string to `double`.
`strtof`[C99]	Convert a number string to `float`.
`strtol`	Convert a number string to `long int`.
`strtold`[C99]	Convert a number string to `long double`.
`strtoul`	Convert a number string to `unsigned long int`.
`strtoll`[C99]	Convert a number string to `long long int`.
`strtoull`[C99]	Convert a number string to `unsigned long long int`.

The `ato*` family of functions has existed for many years. However, the functions have some shortcomings. First, they begin with the letter 'a' to reflect their origin on ASCII-based systems. Second, there is no way for them to report if their input strings contain non-numeric characters, and if so, where the first such character is located. The replacement `strto*` family was invented to solve these problems.

`atoi`, `atol`, and `atoll` treat their input as being decimal, regardless of any leading zero. `strtol`, `strtoul`, `strtoll`, and `strtoull`, however, can be made to recognize numbers of any given base. For more details on these functions, refer to your compiler's library manual.

> **Tip:** When converting strings containing numbers to arithmetic data types, use the `strto*` function family rather than the `ato*` family, since the former is more robust and provides more capability.

3.6 The `sizeof` Operator

The `sizeof` operator is different from all the operators we've seen thus far in two ways: It is spelled using a keyword, and it is evaluated at compile time rather than at runtime, resulting in a compile-time integer constant expression. It can be used in the following two ways:

`sizeof(` *type* `)`

`sizeof` *expression*

When used in the first form, `sizeof` reports the number of bytes needed to store an object of the given type. When used in the second form, `sizeof` reports the number of bytes needed to store an object having the same type as that of *expression*. When the second form is used, the expression is examined at compile time to determine its resulting type; the expression is *not* otherwise evaluated. For example, `sizeof i++` does not actually cause `i` to be incremented. By definition, `sizeof(char)` is 1.

> **Style Tip:** Although the form `sizeof` *expression* does not need to contain parentheses, they often are used anyway, becoming redundant grouping parentheses. That way, you need to remember only one syntax for this operator.

As we saw in §3.4, `memcpy` is used to copy a block of memory from one place to another. In the following example, we use it to copy a whole array of five `int`s; given the definitions of `i1` and `i2`, the following four versions of the copy are equivalent:

```
#include <string.h>

int i1[5], i2[5];

/*1*/ memcpy(i2, i1, 5 * sizeof(int));
/*2*/ memcpy(i2, i1, sizeof(int [5]));
/*3*/ memcpy(i2, i1, sizeof(i2));
/*4*/ memcpy(i2, i1, sizeof(i1));
```

Here, we are using `sizeof` as an abstraction tool, thus avoiding the need to hard-code, or even know, the size of an `int` object.

> **Style Tip:** The use of `sizeof` avoids hard-coding object-type sizes. It provides a level of abstraction that is particularly useful in writing portable code.

`sizeof` can be used very effectively to process every element in an array. For example (see directory ar10):

```
#include <stdio.h>

int main()
{
        unsigned int value[] = {1, 23, 54, 67, 87};
        unsigned int i;

        for (i = 0; i < sizeof(value)/sizeof(value[0]); ++i)
        {
                printf("value[%u] = %2u\n", i, value[i]);
        }

        return 0;
}
```

The output produced is:

```
value[0] =  1
value[1] = 23
value[2] = 54
value[3] = 67
value[4] = 87
```


The compiler figures out the size of the array's dimension from the initializer list each time the program is compiled. If the number of initializers in the list changes, the new size is reflected when the code is re-compiled. The expression:

```
sizeof(value)/sizeof(value[0])
```

is treated by the compiler as follows:

```
(5×sizeof(unsigned int))/sizeof(unsigned int)
```

which results in 5, *regardless* of the representation of an `unsigned int`. That is, the size of an array divided by the size of one of its elements must, by definition, result in the number of elements in that array. The computation applies to arrays of any type and is completely portable. It even works with multidimensional arrays. Remember that while the computation looks expensive, it involves the division of two compile-time constants; and that will typically be done at compile time. (See §6.2.2 for a way to hide this kind of expression behind a preprocessor macro name called `NUMELEM`.)

> **Style Tip:** While `sizeof` does not directly report the number of elements in an array, it can be used to determine that number.

In some applications, the size of one array is proportional to the size of another array or variable. For example, we might have two arrays, both of which must be the same size. Alternatively, perhaps one array must have four times as many elements as another (see directory ar11):

```
void test()
{
/*1*/   char name1[] = "Smith";
/*2*/   char name2[sizeof(name1)];
/*3*/   int counts[20];
/*4*/   double values[sizeof(counts)/sizeof(counts[0]) * 4];
}
```

The size of an array's dimension must be specified using a compile-time integer constant expression. Since that is what is produced by `sizeof`, that operator can be used in this context, as shown above. Since `sizeof(char)` is 1, we do not need to divide by `sizeof(name1[0])` in case 2.

> **Exercise 3-6*:** Define an `int` array, and initialize its elements using the values -20, 52, 33, 0, -5, 74, -6, 1, 9, and -4. Calculate and display the average of the elements as a floating-point value. Make the solution general by not hard-coding the size of the array.
>
> Since the sum of a set of small integers can safely be stored in an integer, store the accumulated sum in an integer variable. In fact, avoid defining any floating-point variables in the program. (See labs directory lbar02.) Here's an example of the output:
>
> `Sum = 134, average = 13.4`
>
> **Exercise 3-7*:** Define and initialize a character array with a string. Define a second character array that is big enough to hold a new string that consists of two copies of the string from the first

array, one concatenated to the other. Use `strcpy` to copy the string from the first array to the second. Then use `strcat` to append a second copy. Make sure that the size of the second array is exactly large enough to hold the new string. (See labs directory lbar07.)

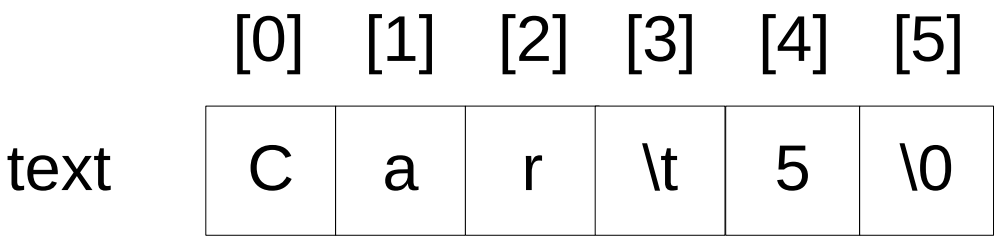
3.7 Character Testing and Conversion

The C library contains functions for testing and converting characters. These are made available via the header `ctype.h`. The following example (see directory ar04) demonstrates several of these functions:

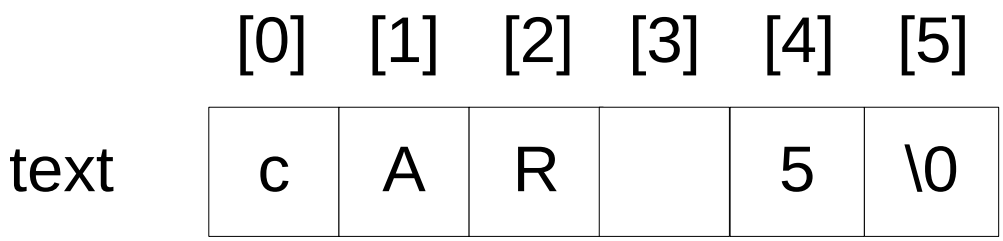
```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main()
{
        char text[] = "Car\t5";
        unsigned int i, length = strlen(text);

        printf("text = %s\n", text);
```




```
        for (i = 0; i < length; ++i)
        {
                if (isupper(text[i]))
                {
                        text[i] = tolower(text[i]);
                }
                else if (islower(text[i]))
                {
                        text[i] = toupper(text[i]);
                }
                else if (isspace(text[i]))
                {
                        text[i] = ' ';
                }
        }
```




```
        printf("text = %s\n", text);

        return 0;
}
```

The output produced is:

```
text = Car      5
text = cAR 5
```

The functions whose names begin with the prefix `is` return a nonzero value[1] if the character passed is in the prescribed category. Otherwise, they return zero.

Note that `tolower` does not actually change the value of its argument. Instead, it returns a value that is the lowercase equivalent of its argument. If the character cannot be converted, the value of the incoming argument is returned. `toupper` works in a similar manner except that it attempts to convert to uppercase instead. The function `isspace` tests to see if the character passed to it is one of the white space characters.

Table 3-4: Character Manipulation Primitives in `ctype.h`

Name	Purpose
`isalnum`	Is character alphanumeric?
`isalpha`	Is character alphabetic?
`isblank`[C99]	Is character blank?
`isctrl`	Is character control?
`isdigit`	Is character a decimal digit?
`isgraph`	Is character graphic?
`islower`	Is character a lowercase alpha?
`isprint`	Is character printable?

[1] Since it is unspecified which nonzero value is used, always check the return value against zero.

Name	Purpose
ispunct	Is character punctuation?
isspace	Is character white space?
isupper	Is character an uppercase alpha?
isxdigit	Is character a hexadecimal digit?
tolower	Convert character to lowercase.
toupper	Convert character to uppercase.

Each function takes a single `int` argument and returns an `int` value.[1]

Implementation-Defined Behavior: The set of printable and control characters for ctype.h.

Clearly, functions such as `isupper` and `isalpha` rely on knowing which alphabet is being used. While English-speaking cultures use the 52 Latin letters for their alphabet, languages such as German, French, Spanish, and Swedish, for example, have other letters. To cater to non-English alphabets, Standard C declares functions whose behavior can legitimately vary from one cultural environment to another, to be *locale-dependent*. By default, these library functions operate in "USA-English" mode.[2]

3.8 Variable-Length Arrays

Prior to C99, arrays had to have dimensions whose size was known at compile time. C99 added the ability to have dimension sizes be integer expressions whose value is not know until runtime, but only for arrays having automatic storage duration (§5.2), which includes automatic variables and parameters. Note, however, that this is a conditional feature; that is, a C99-comformant compiler need not actually support this. Here's how to tell if that feature is available in a C99 implementation:

```
#if __STDC_NO_VLA__ == 1
    /* variable-length arrays are not supported */
#endif
```

Consider the following (see directory ar13):

```
f1(2, 6);

void f1(int m, int n)
{
```

[1] These functions return a value of type `int` rather than `bool` because they were invented long before the `bool` type came into existence.

[2] If you are interested in internationalization, read your compiler's documentation to find more about locales, the standard header `locale.h`, and the `setlocale` function. Also see "Internationalization" in the Advanced-C book.


```
/*1*/    double x1[4];           // fixed-length of 4 elements
/*2*/    double x2[m * 2];       // 2x2=4 elements
/*3*/    double x3[2][n];        // 2 elements, each of which has 6 elements
/*3*/    double x4[m + 2][5];    // 2+2=4 elements, each of which has 5 elements
/*4*/    double x5[n][m];        // 6 elements, each of which has 2 elements
         int i = 3;
/*5*/    double x6[i];           // 3 elements          …
}
```

Consider the following:

```
void f2(int n, int values[n][n*2]);    // OK
void f3(int values[n][n*2], int n);    // error; no n is in scope
```

In the case of `f2`, `n` is seen first, so it can be used in the dimensions for any array parameter following. However, that is not the case for the declaration of `f3`. Then, in the definition of `f2`:

```
void f2(int n, int values[n][n*2]) { … }
```

being a parameter, `values` has automatic storage duration.

Note that we can write the non-definition declaration of `f2`, as follows instead:

```
void f2(int n, int values[*][*]);
```

A type synonym can be created for a variable-length array; for example:

```
int k = 3;
typedef int A[k];    // A is k ints, with k evaluated now, as 3
++k;                 // k = 4
A t1;                // t1 is 3 ints, not 4!
```

A variable-length array definition cannot have an initializer list, and it cannot be a member of a structure or union. And if the runtime size of any dimension is less than 1, the behavior is unspecified. Once the size of a variable-length array has been determined, that size of that array is fixed throughout its life.

7. Pointers and Addresses

So far, we have learned how to do things in C that we already know how to do in some other language. In this chapter, we will be exposed to those capabilities of C that make it different from most high-level languages.

7.1 Introduction

Without a doubt, a difficult part of C to master is pointers. For most high-level language programmers, the idea of using addressing directly and defining variables—called *pointers*—that contain addresses is quite foreign. And even after these programmers come to grips with the idea, they often have considerable difficulty in dealing with the syntax.

To many programmers, pointers seem unnecessary. After all, they have implemented thousands of lines of production code over many years and they have never had the need to even know about, let alone use, pointers. Why then must they master them to use C? That is a reasonable question and one that we will address.

7.2 A Conceptual Model of Memory

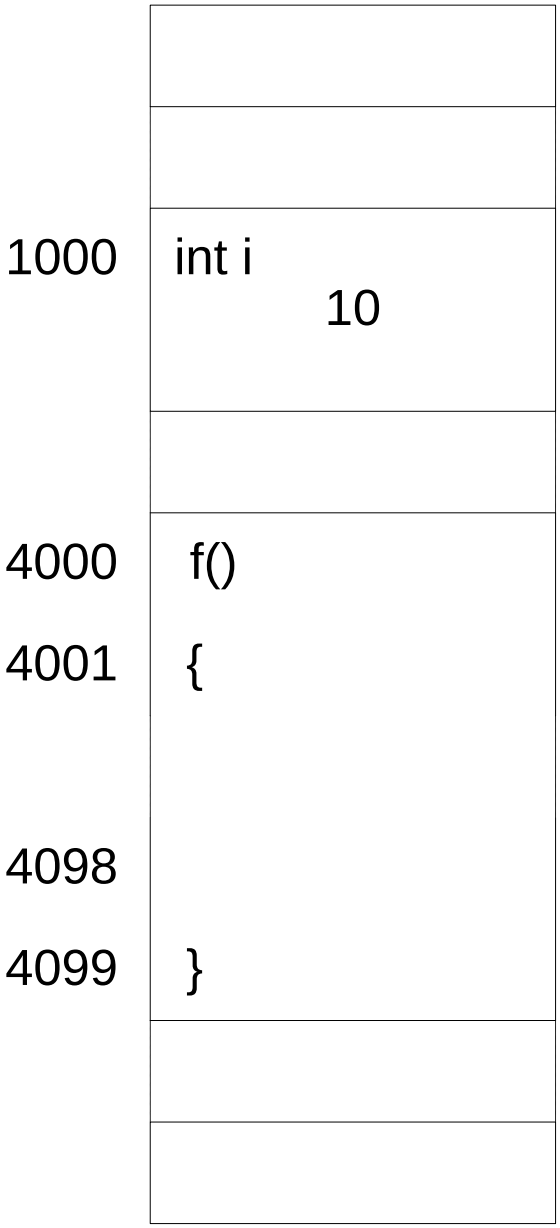
Before we can talk much about addressing and pointers, we need a conceptual model of how memory is organized when a program is loaded for execution. Consider a program that contains the following fragment:

```
void f()
{
        static int i = 10;

        statement(s)
}
```

When this program is loaded into memory for execution, Figure 7–1 below indicates how things might be organized, with regard to the variable `i` and the function `f`. (The exact way in which a system represents this may vary. However, the concepts suggested by this model still apply.)

Figure 7–1: Code and Data Mapping



Memory is organized into an array of cells, with each cell having a unique location, which we shall call an *address*. On machines whose memory is *byte-addressable*, each byte has its own address. On machines that are *word-addressable*, each word has its own address. In the latter case, bytes do not physically exist; characters must be packed into words, and the address of a packed character is the address of its parent word and the character offset within that word.

Many machines use addresses that really are unsigned integers. Others use addresses that are signed integers. Still others use a segmented address that is made up of two parts: a base address and an offset. For the purposes of our discussion, we will make no assumptions about how physical memory really is addressed on any particular machine; that is unimportant in our model.

Function `f` contains a `static int` object, `i`. Being static, this variable typically is allocated memory by the compiler and/or linker. In any event, memory is allocated for it before `main` begins execution. `i` occupies one or

more consecutive locations in memory, starting at address 1000. We say then that the address of `i` is 1000. (The addresses used in this discussion have been picked arbitrarily. For actual memory placement details, use your debugger or consult the listing file produced by your linker.)

Once the compiler and linker have done their job, all variable and function names have disappeared, and all references to them have been reduced to references to their corresponding addresses. And yes, even functions have addresses. In Figure 7–1, the machine instructions for the executable code in function `f` were allocated memory locations 4000–4099.

The reason for all this is that a CPU is a very simple-minded machine. It doesn't know anything about abstractions such as variables and functions. What it does know is how to go to a specific address in memory to get or put a single- or multibyte object. It also knows how to branch or jump to an address and begin executing instructions at that location. Therefore, the principal job of a compiler and linker, in combination, is to reduce source code to an ordered set of machine instructions, many of which deal with addresses.

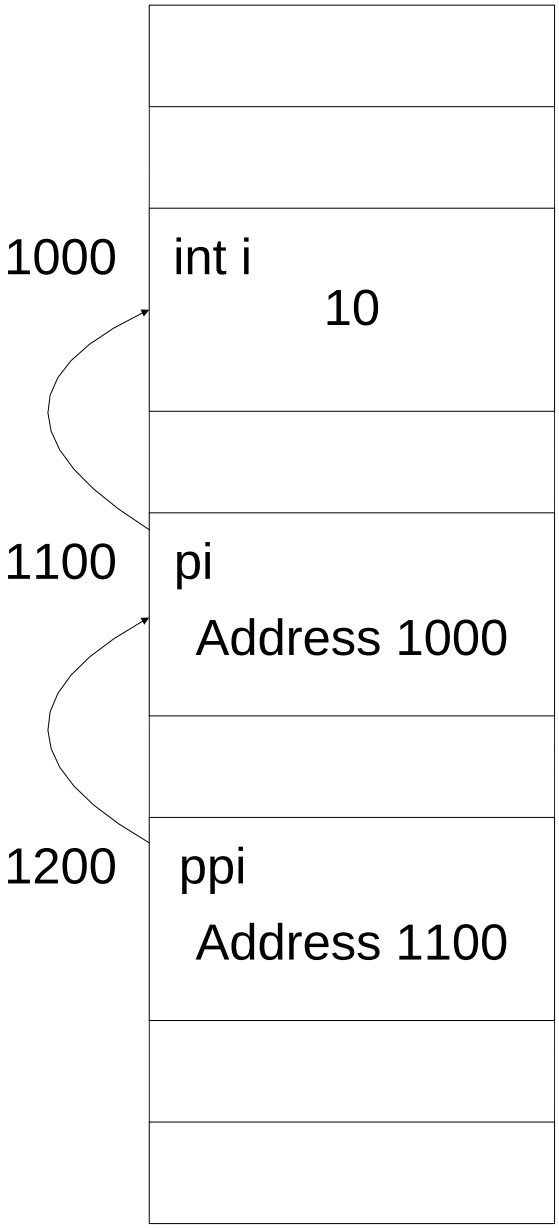
Ultimately, the CPU deals with addresses. However, that doesn't necessarily mean we as programmers need to deal with them. In fact, most languages provide no way to deal with addresses directly. However, C does, and while we can try to avoid using them directly, addresses are produced and used all over the place in things as fundamental as uses of `scanf` and `printf`. There is no way of escaping addresses in C; for better or worse, addresses simply are an integral part of C programming.

As discussed above, objects and functions are referenced via their addresses. And if we change our code in certain ways, when next we compile and link, these objects and functions may be located at different addresses. Therefore, we are rarely, if ever, interested in the particular address at which something is located. In fact, the addresses used during any particular execution of a program are relative to that execution only. Consider the case of an automatic variable. It is allocated memory at run time, typically on some stack. If we call its parent function from several different places, it is almost certain that for each of its lifetimes, the variable will live at different locations. (This would certainly be true for automatic variables defined in functions that are called recursively.)

If addresses can easily change during execution or over the life of an application, why are we interested in them? If we know where something lives, we can remember that information and, later on, access it indirectly through that saved address. Accessing objects and functions in this manner provides a very powerful abstraction tool, and one that we will explore in the next section.

In Figure 7–2, our memory model has been extended to where we can save an object's address. The address of `i` is 1000. If we can find out `i`'s address, we can use it to access `i` indirectly. We might also want to save that address in memory for future reference. We do this by creating a variable capable of storing the address of `i`. Let us call it `pi`, for "pointer to `int`". Let us assume that the pointer variable is located at address 1100.

Figure 7–2: Mapping of Pointers



At this stage, we have a pointer that is initialized with the address of `i`. We say, "`pi` is a pointer with one level of indirection". However, note that `pi` is itself, an object. Therefore, it occupies memory and has a unique address of its own. Therefore, we can take the address of `pi` and store that elsewhere in memory. We do so by storing it in `ppi`, a "pointer to a pointer to an `int`". And as expected, we say that "`ppi` is a pointer with two levels of indirection". Of course, `ppi` has its own address (in this case, let's say 1200), so we can extend the idea of indirection indefinitely. Fortunately, in real-world applications if we need pointers at all, most often we can get by with one level of indirection only. (There are good reasons to use two-level indirection as well. However, it is highly unlikely we will ever need more than two.)

Some machines require or can benefit from *object alignment*. That is, they require or prefer to have objects of certain types aligned on particular address boundaries. Prior to the late 1970s, most machines required almost every C object except those of type `char` to have an address that was at least a multiple of two or four.

Throughout the 1980s, mainstream CPUs were more flexible, for the most part allowing any object to be aligned anywhere in memory. However, optimizing compilers were careful to align objects where possible, since doing so improved memory access times. Late in the 1980s, a new breed of machines became popular. These were based on RISC architectures, and inherent in their design is the requirement that certain object alignment requirements be met.

For the most part, a high-level language programmer need not care about object alignment. However, some knowledge of this subject can be very helpful in understanding C, particularly in debugging *memory access violations*. (On some systems, a memory access violation results in a *core dump*, a display of the contents of certain critical parts of memory—or *core*, as memory used to be called). A memory access violation occurs when a program attempts to access a location in memory for which it has no privilege, or that is not part of the current program. It can also result from an attempt to access an unaligned object on a machine that requires alignment.

Exercise 7-1: Does your hardware have any object alignment requirements? If so, what are they? If not, can it benefit from certain object alignment anyway?

7.3 Pointers as an Abstraction Tool

We have used the term "abstraction" a number of times. Just what is abstraction? As it applies to computing, an *abstraction* is a logical model that allows an understanding of some concept without implying a particular physical implementation. An abstraction is a "black box" that, while we know how to use it or how to interface to it, we do not need to have knowledge of how it works behind the scenes.

Programming involves all kinds of abstraction. For example, an object library allows us to call all kinds of useful routines knowing only their name and argument list attributes. Variable names and functions themselves are abstractions, freeing us from knowing the details of memory allocation and placement. Indeed, languages at a level higher than machine code are a form of abstraction.

C provides a number of important abstraction mechanisms over and above those provided by most high-level languages; examples include type synonyms, enumerated types, and the preprocessor.

There are two primary reasons for having an abstraction: to give a more meaningful name to an obscure or complex constant, expression, or construct; and to allow flexibility so that changing the logical-to-physical mapping requires a change in only one place. Ideally, any such changes should have as little impact on the program as possible—and preferably none. (In fact, that is one of the basic tenets of software reusability and Object-Oriented Design.) Properly used, abstraction helps with consistency and clarity of code and should be viewed as a quality assurance tool. It can also help with portability and provides flexibility when things change over the life of the system.

Tip: A pointer is a tool that can be used to achieve abstraction.

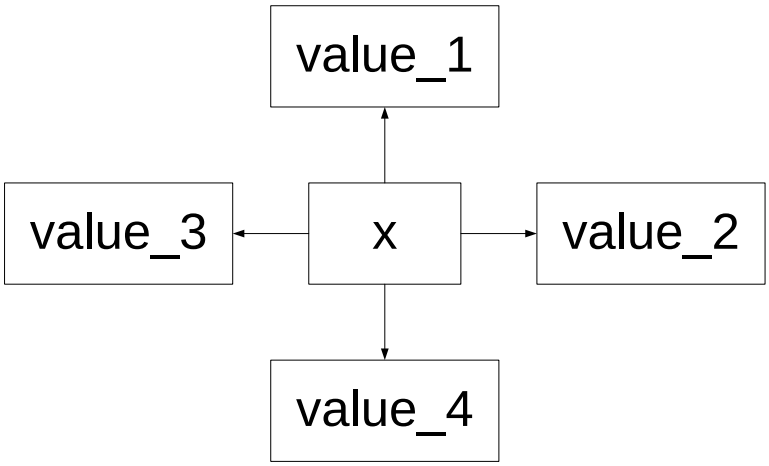
Let us consider a number of problems in which abstraction can be used to make a better or more efficient solution.

7.3.1 Context-Dependent Programs

Many programs need to deal with a number of different states or *contexts*. For example, a decision is made depending on which of a number of menu selections was chosen, a transaction is processed according to its type, or different execution paths are selected depending on whether some condition is true or false.

Let us consider the case in which a program needs to use the value of some coefficient x in many different places. The value used for x depends on the context in which the program executes. For example, in context 1, `x = value_1`; in context 2, `x = value_2`; in context 3, `x = value_3`, and so on. Figure 7–3 shows x and a subset of the context-dependent objects from which it may take its value.

Figure 7–3: Variables are Scattered

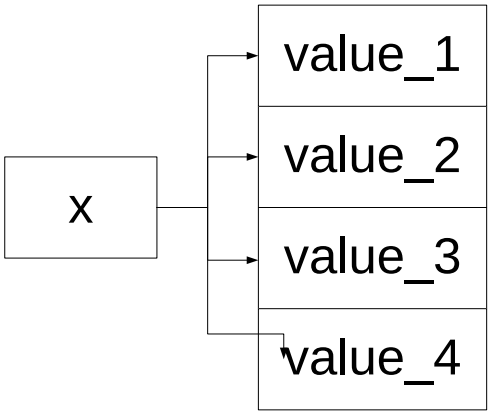


When the context changes, we can set x to be the correct coefficient, as follows:

```
If context-mode = 1
        let x = value_1
Else If context-mode = 2
        let x = value_2
Else If context-mode = 3
        let x = value_3
…
Else
        let x = value_n
```

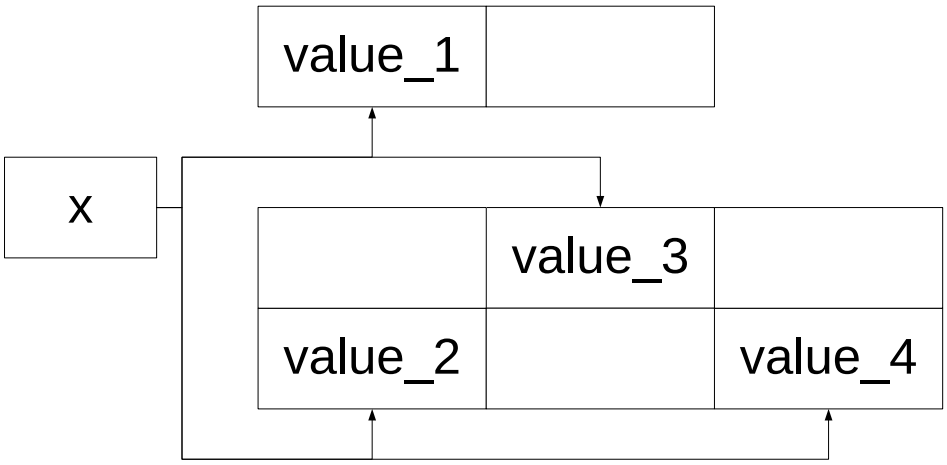
Consider the case in which we wish to update the original source of x. Perhaps we are using an approximation method to solve a problem, and the more computations we do, the better our guess becomes. While x gets us the value we need in any computation, we have no way of knowing where that value came from. If we wish to change the value of the original object, we must revert to the verbose context-dependent test shown above. This test is tedious to read and write, and it generates an amount of code proportional to the number of tests.

Figure 7–4: Variables are Organized



We can solve the problem of changing the original object's value, *provided* we can organize *all* of the context-dependent objects into an array. Now x can become an index into that array, allowing us to both read and write array elements, as shown in Figure 7–4 above. However, it is not always convenient or possible to arrange a set of objects into an array. Perhaps some of the values are in one array or data structure and some are in another, as shown in Figure 7–5. Some may even be scalars. If an object is already in one form of list, it can't physically be part of another list as well.

Figure 7–5: Variables are Scattered



Even if we need only read access to one of some number of context-dependent objects, what happens if the amount of data for each context is nontrivial? For example, each context requires us to use a different array or data structure, each of which contains hundreds or even thousands of values? Copying these data structures takes code and time. Can we even afford to make a private copy? And what happens if one or more of the original objects is updated asynchronously while we are using the copy? We would continue using the old version even though it may be hopelessly out of date.

We have talked only about context-dependent data. It is very likely we also have context-dependent functions. However, functions cannot be copied, nor can they be organized in arrays. So, either way, the solutions proposed above are insufficient.

We can solve all these problems via pointers. When we first detect a change in context, we make a pointer point to the corresponding context, as follows:

```
If context-mode = 1
        let pointer p = address-of value_1
Else If context-mode = 2
        let pointer p = address-of value_2
Else If context-mode = 3
        let pointer p = address-of value_3
...
Else
        let pointer p = address-of value_n
```

Once we have made p point to the corresponding context, p allows us to get at the object to which it points—that is, the underlying object—for both read and write. The underlying object can be of any type and can be arbitrarily large. If the value of the underlying object is changed asynchronously to the main thread of execution, the value we get when we access an object via the pointer is always the current value, since there are no copies of objects in existence.

By storing the address of a function in a pointer, we can use the same technique to call functions indirectly.[1]

7.3.2 Sorting and Searching

A common application involves storing a list of elements in some sorted order, so it can be searched in an efficient manner. Consider the case in which we have a number of personnel records, each containing an employee's last name and employee badge number, among other things. We can represent the employees in a sorted list using an array, as shown in Figure 7–6 (left side). However, in an array, the logical order of the elements must directly correspond to their physical order. This means that the elements can be sorted in only one order at a time. To process the records in badge order when they are stored in name order requires they first be resorted, as shown in Figure 7–6 (right side).

[1] Pointers to functions are discussed in §7.13.

Figure 7–6: Records Sorted by Name (left side) and by Badge (right side)

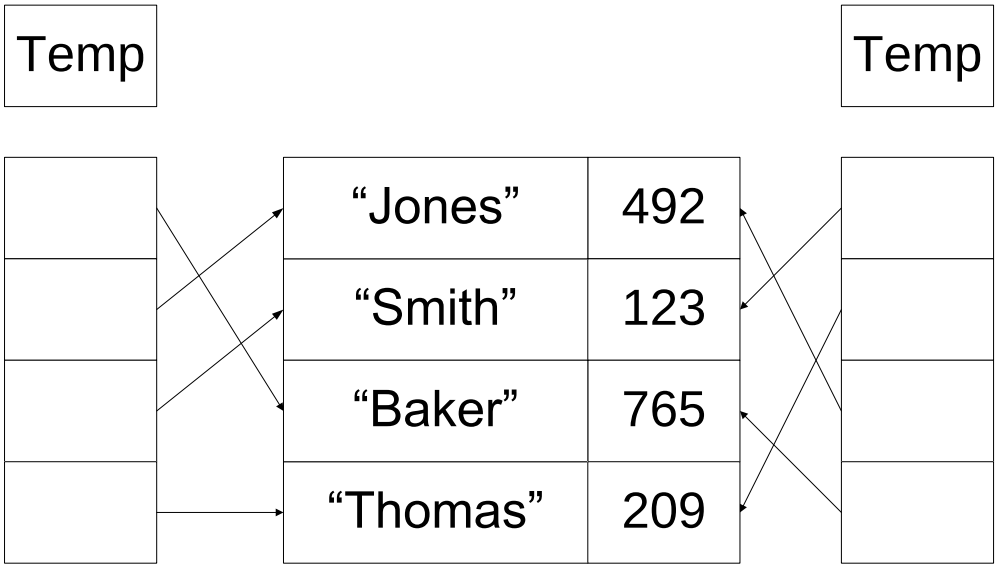
Swap Area	
“Baker”	765
“Jones”	492
“Smith”	123
“Thomas”	209

Swap Area	
“Smith”	123
“Thomas”	209
“Jones”	492
“Baker”	765

It is very inconvenient, and for large arrays inefficient, to continually resort the array at run time. Instead, it is desirable to be able to process the records by any number of orderings without having to pay a run-time cost. This can be achieved by using arrays of pointers, one per sort key.

As shown in Figure 7–7, the records are physically stored in the array in some (possibly random) order. We can sort the records in name order in such a way that if we detect two records need to be swapped over, instead of swapping the records themselves, we swap their corresponding pointers in the key pointer array on the left. Similarly, we reorder the key pointer array on the right for the badge field. We could have key pointer arrays for other data fields as well. Once the pointer arrays have been reordered, we can process a key in ascending order by accessing the corresponding pointer array sequentially. We can also perform a binary search on that array using any of the keys.

Figure 7–7: Sorted Access by Keys



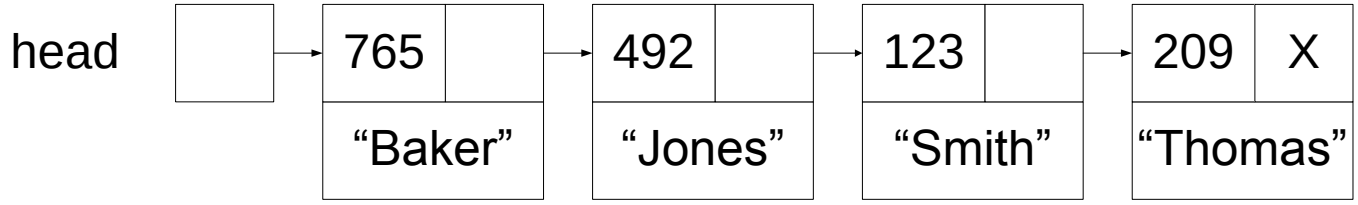
Clearly, the second approach provides a great deal more flexibility. While it requires arrays of pointers to be stored, one per key, it does not require write access to the records themselves, only to the key pointer arrays. It also allows sorting to be faster, since a swap involves swapping two pointers, not two whole records.

In all fairness, the problem as described thus far can be solved without using pointers. If the records are stored in an array, the key pointer arrays could simply contain the indexes into the data array. However, if the records are not organized as an array, we would have to use pointers instead.

7.3.3 Lists

The most common way of representing a list is as an array. By requiring the logical order of elements to match exactly their physical order, arrays lend themselves to random accessing. However, arrays are less useful if elements are inserted and removed from the list at other than the end on a regular basis at run time. An alternate representation that permits efficient insertions and deletions is a *linked list*. Figure 7–8 shows how our employee records can be stored in such a list. Each record not only contains the data for a given employee; it also contains information to allow us to get to its successor. This information is best represented by a pointer. The forward pointer of the last link contains some special value to indicate this is the end of the list. The head pointer points to the first record in the sequence.

Figure 7–8: Singly Linked List



7.4 Using Addresses

Based on our discussion regarding a model of memory, we know that every distinct object and function in a program has a unique address. We also know that arrays are always passed by address. The following example (see directory pt01) shows how we can pass arrays by address explicitly, as well as implicitly:

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

void display(const int values[], unsigned long int count);

int main()
{
        int v1[4], v2[] = {10, 20, 30, 40, 50, 60};

/*1*/   memcpy(v1, v2, 4 * sizeof(int));
        display(v1, 4);                 /*  10  20  30  40 */

/*2*/   memcpy(&v1[0], &v2[0], 4 * sizeof(int));
        display(v1, 4);                 /*  10  20  30  40 */
```

```
/*3*/   memcpy(&v1[1], &v2[3], 3 * sizeof(int));
        display(v1, 4);                 /*  10  40  50  60 */

        return 0;
}

void display(const int values[], unsigned long int count)
{
        unsigned long int i;
        for (i = 0; i < count; ++i)
        {
                printf("%4d", values[i]);
        }
        printf("\n");
}
```

In case 1, `v1` and `v2` are implicitly passed to `memcpy` by address. However, in case 2, we use the *address-of* operator & to take explicitly the address of the first element of each array. While case 2 states explicitly what is already happening in case 1, case 1 is more common, even though the addressing is covert.[1]

The fact that arrays are always passed by address leads to the following, very important rule:

Except for a few notable exceptions (such as sizeof), an expression that designates an array is always converted to the address of its first element. That is,

`array_name` is equivalent to `&array_name[0]`

To take the address of an element other than the first, we must address that element explicitly, as shown in case 3.

Note particularly that we have no interest whatsoever in knowing the precise address of any of these objects. As long as the program can find the intended object at run time, the actual address is irrelevant. And the program is completely portable.

In §3.4 and §3.5, we learned that, unlike many languages, C does not permit arrays to be manipulated as a whole. To copy and compare arrays, for example, we need functions like `memcpy` and `memcmp`. Now that we know about array name conversion, let's examine what happens when we try to copy arrays directly. Assume `a` and `b` are two arrays of the same type and size:

```
a = b /* is treated as &a[0] = &b[0], resulting in a compilation error */
```

Since the address of an object such as `a[0]` is constant for the whole of the object's life, the left-hand operand is a constant, resulting in a compilation error; you can't assign something to a constant!

Consider the following equally likely case in which we attempt to compare two arrays directly:

```
a == b /* is treated as &a[0] == &b[0], which is always false */
```

[1] Don't confuse the unary address-of operator with the binary bitwise AND operator.

Unfortunately, the compiler accepts this, but never produces the (supposedly) expected answer. Since `a` and `b` are distinct objects, their first elements must have different addresses, so the comparison can never be true. The problem is that while the programmer is presumably trying to compare the contents of the two arrays, the compiler has generated code to compare the addresses of their first elements.

Consider the case in which we want to read in from the keyboard, a number of numeric values using `scanf`. In order for `scanf` to be able to store the values read into the variables passed to it, those variables must be passed in by address, and that requires using the address-of operator. For example (see directory pt01b):

```
#include <stdio.h>

int main()
{
        int i;
        float f;
        double d;

        printf("Please enter an int, a float, and a double: ");
/*1*/   scanf("%d %f %lf", &i, &f, &d);
/*2*/   printf("i = %d, f = %f, d = %f\n", i, f, d);

        return 0;
}
```

An example of some input and output is as follows:

```
Please enter an int, a float, and a double: 345 12.34 1.23456
i = 345, f = 12.340000, d = 1.234560
```

Note how in case 1, all three scalar variables have their address taken, and that address is passed to `scanf`, which proceeds to read in values and store them in its arguments in the order it sees the conversion specifiers.

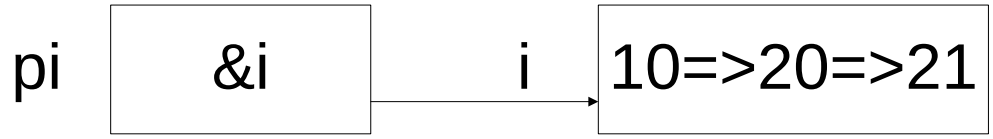
In case 2, we see that the same conversion specifier (`%f`) is used for `float` and `double`. That's because an argument of type `float` is implicitly widened to `double` in the process. However, such widening does not occur on input, so we need different conversion specifiers for these types, `%f` for `float` and `%lf` for `double`.

7.5 Using Pointers

We have learned how to take the address of an object. Now we will see how to save that address in a pointer variable. We'll also learn how to access the object via that pointer's contents (see directory pt02):

```
#include <stdio.h>

int main()
{
        int i = 10;
/*1a*/  int *pi;
/*1b*/  pi = &i;                /* make pi point to i */
```


```
/*2*/   printf("i = %d, *pi = %d\n", i, *pi);

/*3*/   *pi = 20;               /* assign 20 to i */

        printf("i = %d, *pi = %d\n", i, *pi);

/*4*/   ++(*pi);                /* increment i by 1 */

        printf("i = %d, *pi = %d\n", i, *pi);

        return 0;
}
```

The output produced is:

```
i = 10, *pi = 10
i = 20, *pi = 20
i = 21, *pi = 21
```

In case 1a, we define an automatic variable called `pi`, to be a pointer capable of pointing to an object of type `int`. It is the * punctuator preceding `pi`'s name that makes `pi` a pointer. However, what is the *real* type of `pi`? Simply stated, `pi` is an object that is large enough to be able to store the address of any `int` object on the given host system. What `pi` actually looks like internally is of no consequence to us as C programmers. Certainly, we might know or find out the answer for a given implementation, but for the most part, such information is unimportant.

> **Tip:** On many mainstream systems, all addresses are created equal. As a result, many people programming on such machines believe that pointers to all types have exactly the same size and representation. This is not true for all systems and it certainly is not required by C. It is a bad idea to generalize about the representation of pointers.

`pi` is initialized with the address of `i` in case 1b. The type of `pi` is "pointer to `int`". This is also the type of the expression `&i`.

In C, we use the term "pointer" in two ways: to identify a variable that has some pointer type, and to describe the type of an address expression.

> **Tip:** We need not have a pointer variable to have a pointer expression; taking the address of any object produces a pointer expression.

In case 2, we access the value of `i` indirectly through `pi`, by applying the unary indirection operator *. The expression `*pi` translates into "get the address stored in `pi`, go to that address, and get the value of the object

stored at that location". We say that we are *dereferencing* the pointer. When we dereference a pointer of type T we get a result having type T. Therefore, the type of `*pi` is `int`. Not only does the expression `*pi` have type `int`, but it also refers to exactly the same variable as the name `i`.

> **Tip:** Provided a pointer `pi` continues to point to an object `i`, the expressions `i` and `*pi` are synonyms; whatever we can do with one expression, we can do with the other.

Now that we know what the expression `*pi` means, we can revisit the definition of `pi`. By reading the * as the dereferencing operator, we say, "`pi` is an object that when dereferenced, produces an `int` result". From this, we see that, like other variable types, pointers are used just as they are defined.

In case 3, we assign 20 to the `int` pointed to by `pi`.[1] That is, indirectly, we assign 20 to `i`. (Remember, `*pi` *is* `i`.) And in case 4, we increment the `int` pointed to, by 1. That is, we increment `i`. The modifications made in both these cases are reflected in the values output for `i`.

> **Style Tip:** The 'p' prefix on pointer names is this author's attempt at a visual aid to help recognize that these identifiers designate addresses. Since the name of a pointer is an identifier, you may spell pointers names in any valid way you wish.

In §4.2.1, there was an example of passing a scalar by value. We saw that its value could not be changed by the called function. The following version of that program (see directory pt03a) allows the original object's value to be changed:

```
#include <stdio.h>

/*1*/   void init(int *);       /* expects an address */

int main()
{
        int i = 3;

        printf("i = %d\n", i);
/*2*/   init(&i);               /* pass by address */
        printf("i = %d\n", i);

        return 0;
}
```

[1] The notation => is used in pictures to indicate the value taken on by the corresponding object.

```
void init(int *pi)
{
/*3*/   *pi = 5;                /* modify scalar indirectly */
}
```

The output produced is:

```
i = 3
i = 5
```

In case 1, the prototype for `init` indicates that it expects one argument whose type is "pointer to `int`". In case 2, we pass in the address of an `int`. As we learned earlier, it is not necessary to have a pointer variable to have a pointer expression; `&i` *is* a pointer expression, so the function call is well-formed.

Inside the function `init`, the parameter `pi` is a block-scope pointer whose value is the address of `i`. Therefore, in case 3, 5 is assigned to `i`, as shown by the output.

Once pointers are declared, they can point only to objects of the type specified in their declaration. Consider the following example (see directory pt04):

```
int i;
double d;
long int l;
unsigned int u;
int *pi;

pi = &i;                /* OK    */
pi = &d;                /* error */
pi = &l;                /* error */
pi = &u;                /* error */
```

`pi` is declared to point to an object of type `int`. Therefore, it cannot be assigned the address of an object of a different type, not even an `unsigned int`.

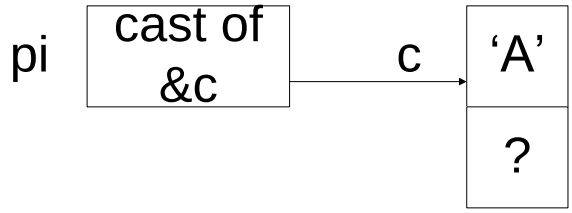
> **Tip:** A pointer type is assignment compatible only with its own pointer type. Therefore, pointers to different types cannot be mixed.

Why do we lock in the underlying type when we declare a pointer? It is insufficient just to declare an object as a pointer. When the CPU goes to the location to which the pointer points, it has to know what kind of object is stored at that location. Should it execute a "move `char`" instruction? A "move `unsigned int`?" A "move `double`?" In assembly language, pointers are typically generic; they can point to anything at any time. Of course, we tell the machine what they really are pointing at when we write the instruction to manipulate that object. In a higher-level language, it is less error-prone to state the underlying type once in the declaration and then never again. Since a compiler knows how a pointer was declared, it always knows how to dereference it.

While pointers of different types are not assignment compatible, one pointer type can be converted explicitly to another, via a cast. However, unless we are involved in some very specialized system programming, we should never need to convert pointer types. In fact, converting from one pointer type to another can result in undefined

behavior, which sometimes manifests itself as a fatal run-time error. Consider the following example (see directory pt05):

```
void f()
{
        char c = 'A';
        int *pi;
        int i;
```




```
/*1*/   pi = (int *)&c;
/*2*/   i = *pi;        /* undefined behavior */
/*3*/   *pi = 10;       /* undefined behavior */
}
```

In case 1, we convert the value of the pointer to `char` expression to type pointer to `int` explicitly. That is, we pretend we are pointing to an `int` when we are not. Without this conversion, the assignment will be rejected. After case 1, `pi` does point to a predictable place; however, it points at a `char` while the compiler thinks it points at an `int`. This makes the behavior of the dereferencing operation in case 2 undefined. Let's consider the possible outcomes. The expression `i = *pi` generates a "move `int`" instruction, and if an `int` is larger than a `char` (which it usually is), this results in the access of memory immediately following that occupied by `c`. And since we have no control over where objects are placed in memory relative to one another, the "`int`" that results from such an access has no predictable or useful value. Case 3 is even more interesting, since it almost certainly results in the overwriting of memory immediately following `c`.

Remember too, that on some systems, objects of certain types need to be properly aligned. On such a system, if the address of `c` is not suitably aligned for use as the address of an `int`, a memory access violation will occur. Consider, for example, a machine that requires `int`s to be aligned on addresses that are a multiple of four. On such a machine, there is a 75 percent chance that the address of `c` is not suitably aligned.

When interfacing to memory-mapped hardware, sometimes we must initialize a pointer to an absolute address. We can do this by converting the address via an explicit cast, as follows:

```
void f()
{
        int *pi = (int *)0xFFFE;

        *pi = 100;
}
```

One common application of this technique is to write directly to a video display or a device register.

> **Tip:** Don't play fast and loose with pointer-to-pointer, pointer-to-integer, and integer-to-pointer conversions.

In our conceptual memory model, not only did we have a pointer to an `int`, but we also took that pointer's address and stored it in a pointer to a pointer to an `int`, called `ppi`. The following example (see directory pt06) defines the objects from that model, and initializes and dereferences them appropriately:

```
#include <stdio.h>

int main()
{
        int i = 10;
/*1*/   int *pi = &i;           /* pi is a ptr to int         */
/*2*/   int **ppi = &pi;        /* ppi is a ptr to ptr to int */
```


```
/*3*/   printf("i = %d, *pi = %d, **ppi = %d\n", i, *pi, **ppi);

        return 0;
}
```

The output produced is:

```
i = 10, *pi = 10, **ppi = 10
```

In a pointer declaration, we use one prefix * punctuator per level of indirection. In case 1, we define a pointer to one level, so we use one * only. In case 2, we want `ppi` to be a pointer to two levels, so we use **. (There is no limit to the number of levels of indirection permitted.)

In case 3, we dereference each pointer using the notation corresponding to the result we wish to obtain.

One of the biggest problems programmers encounter when learning pointer notation is when to use the * operator and how many of them to use. The following table contains a number of declarations, the set of expressions that can arise from those declarations, and the type of each expression. As we shall see, the way we use an identifier in an expression corresponds exactly to the way we declared it:

7-1: Declarations and Their Usage:

Declaration	Expression	Type
`int i`	`i`	`int ~~i~~`
`int i[5]`	`i`	`int ~~i~~[5]`
	`i[1]`	`int ~~i[5]~~`

Declaration	Expression	Type
`int f()`	`f`	`int ~~f~~()`
	`f()`	`int ~~f()~~`
`int *pi`	`pi`	`int *~~pi~~`
	`*pi`	`int ~~*pi~~`
`int **ppi`	`ppi`	`int **~~ppi~~`
	`*ppi`	`int *~~*ppi~~`
	`**ppi`	`int ~~**ppi~~`

From a reliability viewpoint, we must be able to tell, at a glance, the type of every expression we read or write. To do this, look at the expression in question, and then hide in the original declaration all the tokens that exist in that expression. The tokens that remain visible make up the type of that expression. For example, in the table above, `int i[5]` declares `i` to be "an array of five `int`". For the expression `i`, we hide the token `i` in the declaration, which results in the type "array of five `int`". By applying the same approach to `pi` and `ppi`, we can deduce that the types of the expression `*pi` and `*ppi` are `int` and `int *`, respectively. It really is that easy.

> **Style Tip:** We have seen the 'p' prefix on the names of pointers to one level of indirection. By adding one 'p' prefix for each level of indirection, we can encode even more meaning into the name.

For the most part, we can get by using one level of indirection only. However, on occasion, it is useful to use two. In the following program (see directory pt07a), we pass a pointer to `char` to a function by address so that pointer can be modified by the function:

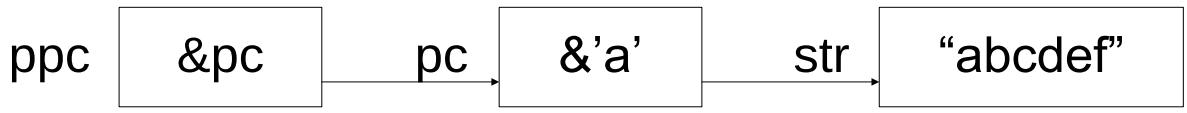
```
#include <stdio.h>

/*1*/   void setup(char **);

int main()
{
        char *pc;

/*2*/   setup(&pc);
/*3*/   printf("text = %s\n", pc);

        return 0;
}
```




```
void setup(char **ppc)
{
/*4*/   static char str[] = "abcdef";

/*5*/   *ppc = str;
}
```

The output produced is:

```
text = abcdef
```

In case 1, `setup` expects one argument, of type "pointer to pointer to `char`". We construct an expression of this type by taking the address of `pc` in case 2. Assuming that `pc` is initialized properly inside `setup`, `printf` procedes to display the text to which `pc` points, in case 3.

In case 5, we initialize `pc` using the expression `*ppc`. According to the technique we learned in the declaration and usage table above, the type of `*ppc` is "pointer to `char`". `*ppc` is a synonym for `pc`.

Note that in case 4, `str` is static. This is necessary since it makes this array permanent. If it were automatic instead, we would be initializing `pc` to point to something that no longer existed once we returned from `setup`.

> **Tip:** Never export the address of an automatic object to a function that calls that object's parent function. Remember, an automatic object's life ends at the end of its parent block.

7.5.1 Pointers and `const`

The qualifier `const` can be very useful when used in conjunction with data pointers. A data pointer declaration actually declares the attributes of two things: the pointer itself and the object to which it points. Since both the pointer and the object being pointed at might be `const`-qualified, `const` can appear more than once in a pointer declaration as shown in the next example (pt08):

```
/* 1*/        char ncc = 'A';
/* 2*/  const char cc  = 'Z';
/* 3*/        char *       ncpncc = &ncc;
/* 4*/  const char *       ncpcc  = &cc;
/* 5*/        char * const cpncc  = &ncc;
/* 6*/  const char * const cpcc   = &cc;
/* 7*/  ncpncc = &cc;           /* error */
/* 8*/   cpncc = &cc;           /* error */
/* 9*/  *ncpcc = 'x';           /* error */
/*10*/   *cpcc = 'x';           /* error */
}
```

In case 1, `ncc` is a non-`const` `char` while, in case 2, `cc` is a `const` `char`. That is, the compiler will allow us to modify `ncc` but not `cc`.

In case 3, ncpncc is a non-const pointer to a non-const char. That is, the compiler will allow us to modify both the pointer ncpncc and the char to which it points.

In case 4, ncpcc is a non-const pointer to a const char. That is, the compiler will allow us to modify the pointer ncpcc but not the char to which it points.

In case 5, cpncc is a const pointer to a non-const char. That is, the compiler will not allow us to modify the pointer cpncc but will allow us to modify the char to which it points.

In case 6, cpcc is a const pointer to a const char. That is, the compiler will not allow us to modify the pointer cpcc or the char to which it points.

Case 7 is diagnosed because we are attempting to store the address of a const-qualified object in a pointer that does not preserve that const protection. Consider the case where cc really is stored in some physically write-protected location in memory. If case 7 were permitted, any attempt to modify the object to which ncpncc points would go undetected at compile-time yet would fail at runtime. The same situation exists in case 8.

In cases 9 and 10, the type of each pointer indicates they point to const-qualified objects. Therefore, the compiler diagnoses any attempt to modify the objects to which they point.

In short, when const is used correctly, the compiler can detect any attempts to directly or indirectly modify a const-qualified object. However, const should be viewed as a quality assurance tool rather than one that enforces security.

Consider the following example (see directory pt09):

```
void f()
{
        char c = 'A';
/*1*/         char *pncc = &c;
/*2*/   const char *pcc  = &c;

/*3*/   c = 'B';        /* OK    */
/*4*/   *pncc = 'B';    /* OK    */
/*5*/   *pcc  = 'B';    /* error */
}
```

Since c is a non-const char, its address can be stored in pncc, a pointer to a non-const char. And, we can modify c directly, as in case 3, or indirectly, as in case 4.

Case 2 is interesting. Here we assign the address of a non-const char to a pointer to a const char. While c actually is modifiable, by assigning its address to pcc, we prohibit c from being modified through pcc. That is, we have switched on read-only protection resulting in case 5 being diagnosed by the compiler. A similar situation exists in the following example (pt10):


```
void g1(      int *);
void g2(const int *);

void f()
{
        int i = 10;

        g1(&i);         /* give g1 read/write access to i */
        g2(&i);         /* give g2 read       access to i */
}
```

Clearly, `i` can be modified directly within function `f`. It can also be modified indirectly in function `g1` since the argument expected by `g1` is a pointer to a non-`const` object. By declaring `g2` to take a pointer to a `const`-qualified object, we prohibit `g2`, and any functions to which it passes the address of its argument, from indirectly modifying `i`.

The ability to make modifiable objects read-only across user-defined parts of a program is very powerful and should be exploited to the fullest. It certainly is used a great deal in the library. Here are some examples of prototypes from various standard headers:

```
char *strcpy(char *__dest, const char *__source);
int strcmp(const char *__string1, const char *__string2);
FILE *fopen(const char *__filename, const char *__mode);
double strtod(const char *__nptr, char **__endptr);
```

Each prototype declares at least one argument of type "pointer to `const` char". However, this does not require that the strings passed to them be `const`-qualified. What it does guarantee is that these functions will not modify those strings, allowing the compiler (potentially) to generate more optimum code when such a function is called in the presence of the appropriate prototype.

Consider the following example (pt11):

```
void f()
{
/*1*/         char *pc1 = "text";
/*2*/   const char *pc2 = "text";

/*3*/   *pc1 = 'x';     /* possible runtime error */
/*4*/   *pc2 = 'x';     /* compile-time error     */
}
```

Since it is undefined as to whether a compiler stores string literals in read-only or read/write memory, the declaration in case 1 allows case 3 to compile without error even though the assignment might fail at run time. The declaration in case 2 causes case 4 to fail at compile time. Therefore, case 2 is preferred over case 1.

7.5.2 Pointers and `restrict`

C99 added the type qualifier `restrict`, which is used in the context of pointers. It is discussed in the Advanced-C book.