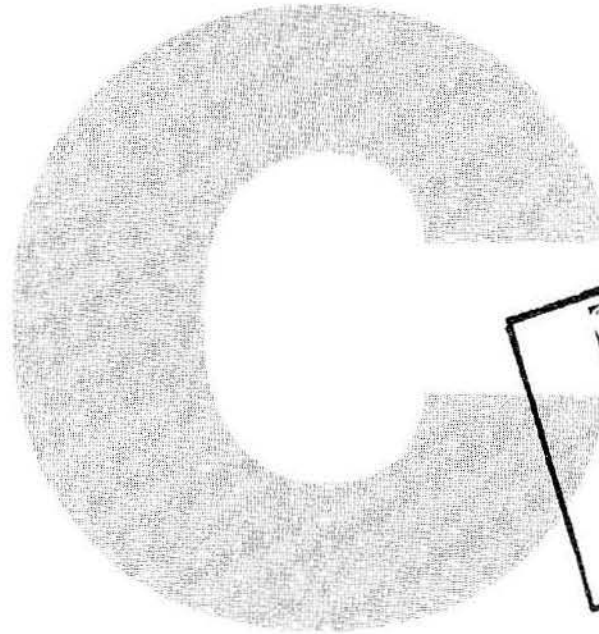


SECOND EDITION

THE



PROGRAMMING
LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

001

ServiceNow's Exhibit No. 1004

Library of Congress Cataloging-in-Publication Data

Kernighan, Brian W.

The C programming language.

Includes index.

**1. C (Computer program language) I. Ritchie,
Dennis M. II. Title.**

QA76.73.C15K47 1988 005.13'3 88-5934

ISBN 0-13-110370-9

ISBN 0-13-110362-8 (pbk.)

Copyright © 1988, 1978 by Bell Telephone Laboratories, Incorporated.



**© Published by Prentice Hall P T R
Prentice-Hall, Inc.
Upper Saddle River, NJ 07458**

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

UNIX is a registered trademark of AT&T.

This book was typeset (picitblieqnitroff -ms) in Times Roman and Courier by the authors, using an Autologic APS-5 phototypesetter and a DEC VAX 8550 running the 9th Edition of the UNIX® operating system.

Prentice Hall Software Series

Brian Kernighan, Advisor

ISBN 0-13-110362-8

**Text printed in the United States on recycled paper at Courier in Westford,
Massachusetts.**

Fifty-Second Printing, February 2014

ISBN 0-13-110362-8 {PBK}

ISBN 0-13-110370-9

Prentice-Hall International (UK) Limited, London

Prentice-Hall of Australia Pty. Limited, Sydney

Prentice-Hall of Canada, Inc., Toronto

Prentice-Hall Hispanoamericana, S. A., Mexico

Prentice-Hall of India Private Limited, New Delhi

Prentice-Hall of Japan, Inc., Tokyo

Prentice-Hall Asia Pte. Ltd., Singapore

Editora Prentice-Hall do Brasil, Ltda., Rio de Janeiro

Contents

Preface	ix
Preface to the First Edition	xi
Introduction	1
Chapter 1. A Tutorial Introduction	5
1.1 Getting Started	5
1.2 Variables and Arithmetic Expressions	8
1.3 The For Statement	13
1.4 Symbolic Constants	14
1.5 Character Input and Output	15
1.6 Arrays	22
1.7 Functions	24
1.8 Arguments—Call by Value	27
1.9 Character Arrays	28
1.10 External Variables and Scope	31
Chapter 2. Types, Operators, and Expressions	35
2.1 Variable Names	35
2.2 Data Types and Sizes	36
2.3 Constants	37
2.4 Declarations	40
2.5 Arithmetic Operators	41
2.6 Relational and Logical Operators	41
2.7 Type Conversions	42
2.8 Increment and Decrement Operators	46
2.9 Bitwise Operators	48
2.10 Assignment Operators and Expressions	50
2.11 Conditional Expressions	51
2.12 Precedence and Order of Evaluation	52
Chapter 3. Control Flow	55
3.1 Statements and Blocks	55
3.2 If-Else	55

3.3	Else-If	57
3.4	Switch	58
3.5	Loops—While and For	60
3.6	Loops—Do-while	63
3.7	Break and Continue	64
3.8	Goto and Labels	65
Chapter 4.	Functions and Program Structure	67
4.1	Basics of Functions	67
4.2	Functions Returning Non-integers	71
4.3	External Variables	73
4.4	Scope Rules	80
4.5	Header Files	81
4.6	Static Variables	83
4.7	Register Variables	83
4.8	Block Structure	84
4.9	Initialization	85
4.10	Recursion	86
4.11	The C Preprocessor	88
Chapter 5.	Pointers and Arrays	93
5.1	Pointers and Addresses	93
5.2	Pointers and Function Arguments	95
5.3	Pointers and Arrays	97
5.4	Address Arithmetic	100
5.5	Character Pointers and Functions	104
5.6	Pointer Arrays; Pointers to Pointers	107
5.7	Multi-dimensional Arrays	110
5.8	Initialization of Pointer Arrays	113
5.9	Pointers vs. Multi-dimensional Arrays	113
5.10	Command-line Arguments	114
5.11	Pointers to Functions	118
5.12	Complicated Declarations	122
Chapter 6.	Structures	127
6.1	Basics of Structures	127
6.2	Structures and Functions	129
6.3	Arrays of Structures	132
6.4	Pointers to Structures	136
6.5	Self-referential Structures	139
6.6	Table Lookup	143
6.7	Typedef	146
6.8	Unions	147
6.9	Bit-fields	149
Chapter 7.	Input and Output	151
7.1	Standard Input and Output	151
7.2	Formatted Output—Printf	153

7.3	Variable-length Argument Lists	155
7.4	Formatted Input—Scanf	157
7.5	File Access	160
7.6	Error Handling—Stderr and Exit	163
7.7	Line Input and Output	164
7.8	Miscellaneous Functions	166
Chapter 8.	The UNIX System Interface	169
8.1	File Descriptors	169
8.2	Low Level I/O—Read and Write	170
8.3	Open, Creat, Close, Unlink	172
8.4	Random Access—Lseek	174
8.5	Example—An Implementation of Fopen and Getc	175
8.6	Example—Listing Directories	179
8.7	Example—A Storage Allocator	185
Appendix A.	Reference Manual	191
A1	Introduction	191
A2	Lexical Conventions	191
A3	Syntax Notation	194
A4	Meaning of Identifiers	195
A5	Objects and Lvalues	197
A6	Conversions	197
A7	Expressions	200
A8	Declarations	210
A9	Statements	222
A10	External Declarations	225
A11	Scope and Linkage	227
A12	Preprocessing	228
A13	Grammar	234
Appendix B.	Standard Library	241
B1	Input and Output: <stdio.h>	241
B2	Character Class Tests: <ctype.h>	248
B3	String Functions: <string.h>	249
B4	Mathematical Functions: <math.h>	250
B5	Utility Functions: <stdlib.h>	251
B6	Diagnostics: <assert.h>	253
B7	Variable Argument Lists: <stdarg.h>	254
B8	Non-local Jumps: <setjmp.h>	254
B9	Signals: <signal.h>	255
B10	Date and Time Functions: <time.h>	255
B11	Implementation-defined Limits: <limits.h> and <float.h>	257
Appendix C.	Summary of Changes	259
Index		263

Introduction

C is a general-purpose programming language. It has been closely associated with the UNIX system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a "system programming language" because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 for the first UNIX system on the DEC PDP-7.

BCPL and B are "typeless" languages. By contrast, C provides a variety of data types. The fundamental types are characters, and integers and floating-point numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures, and unions. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement. Pointers provide for machine-independent address arithmetic.

C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (`if-else`), selecting one of a set of possible cases (`switch`), looping with the termination test at the top (`while`, `for`) or at the bottom (`do`), and early loop exit (`break`).

Functions may return values of basic types, structures, unions, or pointers. Any function may be called recursively. Local variables are typically "automatic," or created anew with each invocation. Function definitions may not be nested but variables may be declared in a block-structured fashion. The functions of a C program may exist in separate source files that are compiled separately. Variables may be internal to a function, external but known only within a single source file, or visible to the entire program.

A preprocessing step performs macro substitution on program text, inclusion of other source files, and conditional compilation.

C is a relatively "low level" language. This characterization is not

pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

C provides no operations to deal directly with composite objects such as character strings, sets, lists, or arrays. There are no operations that manipulate an entire array or string, although structures may be copied as a unit. The language does not define any storage allocation facility other than static definition and the stack discipline provided by the local variables of functions; there is no heap or garbage collection. Finally, C itself provides no input/output facilities; there are no READ or WRITE statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly-called functions. Most C implementations have included a reasonably standard collection of such functions.

Similarly, C offers only straightforward, single-thread control flow: tests, loops, grouping, and subprograms, but not multiprogramming, parallel operations, synchronization, or coroutines.

Although the absence of some of these features may seem like a grave deficiency ("You mean I have to call a function to compare two character strings?"), keeping the language down to modest size has real benefits. Since C is relatively small, it can be described in a small space, and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language.

For many years, the definition of C was the reference manual in the first edition of *The C Programming Language*. In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C," was completed late in 1988. Most of the features of the standard are already supported by modern compilers.

The standard is based on the original reference manual. The language is relatively little changed; one of the goals of the standard was to make sure that most existing programs would remain valid, or, failing that, that compilers could produce warnings of new behavior.

For most programmers, the most important change is a new syntax for declaring and defining functions. A function declaration can now include a description of the arguments of the function; the definition syntax changes to match. This extra information makes it much easier for compilers to detect errors caused by mismatched arguments; in our experience, it is a very useful addition to the language.

There are other small-scale language changes. Structure assignment and enumerations, which had been widely available, are now officially part of the language. Floating-point computations may now be done in single precision. The properties of arithmetic, especially for unsigned types, are clarified. The preprocessor is more elaborate. Most of these changes will have only minor

effects on most programmers.

A second significant contribution of the standard is the definition of a library to accompany C. It specifies functions for accessing the operating system (for instance, to read and write files), formatted input and output, memory allocation, string manipulation, and the like. A collection of standard headers provides uniform access to declarations of functions and data types. Programs that use this library to interact with a host system are assured of compatible behavior. Most of the library is closely modeled on the "standard I/O library" of the UNIX system. This library was described in the first edition, and has been widely used on other systems as well. Again, most programmers will not see much change.

Because the data types and control structures provided by C are supported directly by most computers, the run-time library required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can be avoided if they are not needed. Most can be written in C, and except for the operating system details they conceal, are themselves portable.

Although C matches the capabilities of many computers, it is independent of any particular machine architecture. With a little care it is easy to write portable programs, that is, programs that can be run without change on a variety of hardware. The standard makes portability issues explicit, and prescribes a set of constants that characterize the machine on which the program is run.

C is not a strongly-typed language, but as it has evolved, its type-checking has been strengthened. The original definition of C frowned on, but permitted, the interchange of pointers and integers; this has long since been eliminated, and the standard now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new function declarations are another step in this direction. Compilers will warn of most type errors, and there is no automatic conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.

C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better. Nonetheless, C has proven to be an extremely effective and expressive language for a wide variety of programming applications.

The book is organized as follows. Chapter 1 is a tutorial on the central part of C. The purpose is to get the reader started as quickly as possible, since we believe strongly that the way to learn a new language is to write programs in it. The tutorial does assume a working knowledge of the basic elements of programming; there is no explanation of computers, of compilation, nor of the meaning of an expression like $n=n+1$. Although we have tried where possible to show useful programming techniques, the book is not intended to be a reference work on data structures and algorithms; when forced to make a choice, we have concentrated on the language.

Chapters 2 through 6 discuss various aspects of C in more detail, and rather more formally, than does Chapter 1, although the emphasis is still on examples of complete programs, rather than isolated fragments. Chapter 2 deals with the basic data types, operators and expressions. Chapter 3 treats control flow: `if-else`, `switch`, `while`, `for`, etc. Chapter 4 covers functions and program structure—external variables, scope rules, multiple source files, and so on—and also touches on the preprocessor. Chapter 5 discusses pointers and address arithmetic. Chapter 6 covers structures and unions.

Chapter 7 describes the standard library, which provides a common interface to the operating system. This library is defined by the ANSI standard and is meant to be supported on all machines that support C, so programs that use it for input, output, and other operating system access can be moved from one system to another without change.

Chapter 8 describes an interface between C programs and the UNIX operating system, concentrating on input/output, the file system, and storage allocation. Although some of this chapter is specific to UNIX systems, programmers who use other systems should still find useful material here, including some insight into how one version of the standard library is implemented, and suggestions on portability.

Appendix A contains a language reference manual. The official statement of the syntax and semantics of C is the ANSI standard itself. That document, however, is intended foremost for compiler writers. The reference manual here conveys the definition of the language more concisely and without the same legalistic style. Appendix B is a summary of the standard library, again for users rather than implementers. Appendix C is a short summary of changes from the original language. In cases of doubt, however, the standard and one's own compiler remain the final authorities on the language.

CHAPTER 5: **Pointers and Arrays**

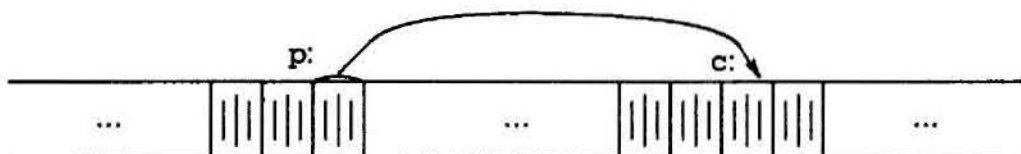
A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the `goto` statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type `void *` (pointer to void) replaces `char *` as the proper type for a generic pointer.

5.1 **Pointers and Addresses**

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a `char`, a pair of one-byte cells can be treated as a `short` integer, and four adjacent bytes form a `long`. A pointer is a group of cells (often two or four) that can hold an address. So if `c` is a `char` and `p` is a pointer that points to it, we could represent the situation this way:



The unary operator `&` gives the address of an object, so the statement

```
p = &c;
```

assigns the address of `c` to the variable `p`, and `p` is said to “point to” `c`. The `&` operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

The unary operator `*` is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that `x` and `y` are integers and `ip` is a pointer to `int`. This artificial sequence shows how to declare a pointer and how to use `&` and `*`:

```
int x = 1, y = 2, z[10];
int *ip;           /* ip is a pointer to int */

ip = &x;           /* ip now points to x */
y = *ip;           /* y is now 1 */
*ip = 0;           /* x is now 0 */
ip = &z[0];        /* ip now points to z[0] */
```

The declarations of `x`, `y`, and `z` are what we’ve seen all along. The declaration of the pointer `ip`,

```
int *ip;
```

is intended as a mnemonic; it says that the expression `*ip` is an `int`. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression `*dp` and `atof(s)` have values of type `double`, and that the argument of `atof` is a pointer to `char`.

You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. (There is one exception: a “pointer to `void`” is used to hold any type of pointer but cannot be dereferenced itself. We’ll come back to it in Section 5.11.)

If `ip` points to the integer `x`, then `*ip` can occur in any context where `x` could, so

```
*ip = *ip + 10;
```

increments `*ip` by 10.

The unary operators `*` and `&` bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever `ip` points at, adds 1, and assigns the result to `y`, while

```
*ip += 1
```

increments what `ip` points to, as do

```
+++ip
```

and

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment `ip` instead of what it points to, because unary operators like `*` and `++` associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if `iq` is another pointer to `int`,

```
iq = ip
```

copies the contents of `ip` into `iq`, thus making `iq` point to whatever `ip` pointed to.

5.2 Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order elements with a function called `swap`. It is not enough to write

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

where the `swap` function is defined as

```
void swap(int x, int y) /* WRONG */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Because of call by value, `swap` can't affect the arguments `a` and `b` in the routine that called it. The function above only swaps *copies* of `a` and `b`.

The way to obtain the desired effect is for the calling program to pass *pointers* to the values to be changed:

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

Since the operator `&` produces the address of a variable, `&a` is a pointer to `a`. In `swap` itself, the parameters are declared to be pointers, and the operands are accessed indirectly through them.

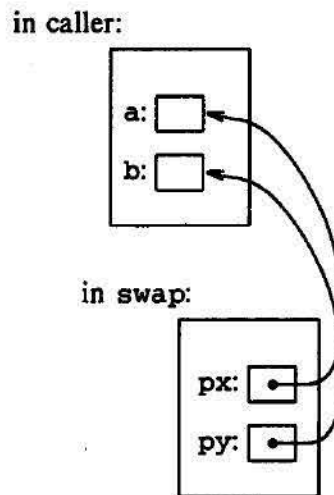

```

void swap(int *px, int *py) /* interchange *px and *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}

```

Pictorially:



Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function `getint` that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for EOF, that could also be the value of an input integer.

One solution is to have `getint` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by `scanf` as well; see Section 7.4.

The following loop fills an array with integers by calls to `getint`:

```

int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
    ;

```

Each call sets `array[n]` to the next integer found in the input and increments `n`. Notice that it is essential to pass the address of `array[n]` to `getint`. Otherwise there is no way for `getint` to communicate the converted integer back to the caller.

Our version of `getint` returns EOF for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```

#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint:  get next integer from input into *pn */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch()))    /* skip white space */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c);    /* it's not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}

```

Throughout `getint`, `*pn` is used as an ordinary `int` variable. We have also used `getch` and `ungetch` (described in Section 4.3) so the one extra character that must be read can be pushed back onto the input.

Exercise 5-1. As written, `getint` treats a `+` or `-` not followed by a digit as a valid representation of zero. Fix it to push such a character back on the input. □

Exercise 5-2. Write `getfloat`, the floating-point analog of `getint`. What type does `getfloat` return as its function value? □

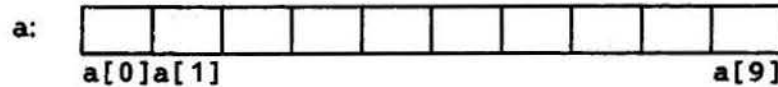
5.3 Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

The declaration

```
int a[10];
```

defines an array *a* of size 10, that is, a block of 10 consecutive objects named *a*[0], *a*[1], ..., *a*[9].



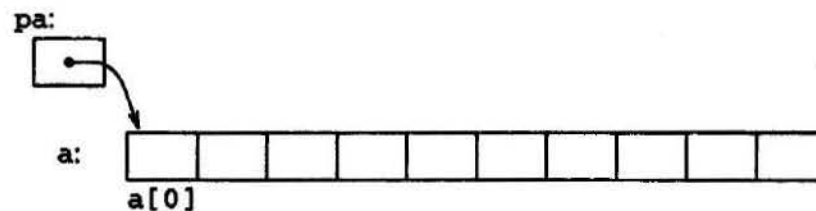
The notation *a*[*i*] refers to the *i*-th element of the array. If *pa* is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets *pa* to point to element zero of *a*; that is, *pa* contains the address of *a*[0].



Now the assignment

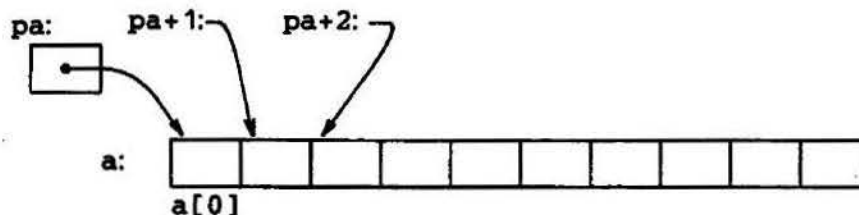
```
x = *pa;
```

will copy the contents of *a*[0] into *x*.

If *pa* points to a particular element of an array, then by definition *pa*+1 points to the next element, *pa*+*i* points *i* elements after *pa*, and *pa*-*i* points *i* elements before. Thus, if *pa* points to *a*[0],

```
*(pa+1)
```

refers to the contents of *a*[1], *pa*+*i* is the address of *a*[*i*], and **(pa*+*i*) is the contents of *a*[*i*].



These remarks are true regardless of the type or size of the variables in the array *a*. The meaning of “adding 1 to a pointer,” and by extension, all pointer arithmetic, is that *pa*+1 points to the next object, and *pa*+*i* points to the *i*-th

object beyond `pa`.

The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

```
pa = &a[0];
```

`pa` and `a` have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment `pa=&a[0]` can also be written as

```
pa = a;
```

Rather more surprising, at least at first sight, is the fact that a reference to `a[i]` can also be written as `*(a+i)`. In evaluating `a[i]`, C converts it to `*(a+i)` immediately; the two forms are equivalent. Applying the operator `&` to both parts of this equivalence, it follows that `&a[i]` and `a+i` are also identical: `a+i` is the address of the *i*-th element beyond `a`. As the other side of this coin, if `pa` is a pointer, expressions may use it with a subscript; `pa[i]` is identical to `*(pa+i)`. In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so `pa=a` and `pa++` are legal. But an array name is not a variable; constructions like `a=pa` and `a++` are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of `strlen`, which computes the length of a string.

```
/* strlen: return length of string s */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

Since `s` is a pointer, incrementing it is perfectly legal; `s++` has no effect on the character string in the function that called `strlen`, but merely increments `strlen`'s private copy of the pointer. That means that calls like

```
strlen("hello, world"); /* string constant */
strlen(array);           /* char array[100]; */
strlen(ptr);             /* char *ptr; */
```

all work.

As formal parameters in a function definition,

```
char s[];
```

and

```
char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the parameter is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if *a* is an array,

```
f(&a[2])
```

and

```
f(a+2)
```

both pass to the function *f* the address of the subarray that starts at *a*[2]. Within *f*, the parameter declaration can read

```
f(int arr[]) { ... }
```

or

```
f(int *arr) { ... }
```

So as far as *f* is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

If one is sure that the elements exist, it is also possible to index backwards in an array; *p*[-1], *p*[-2], and so on are syntactically legal, and refer to the elements that immediately precede *p*[0]. Of course, it is illegal to refer to objects that are not within the array bounds.

5.4 Address Arithmetic

If *p* is a pointer to some element of an array, then *p*++ increments *p* to point to the next element, and *p*+=*i* increments it to point *i* elements beyond where it currently does. These and similar constructions are the simplest forms of pointer or address arithmetic.

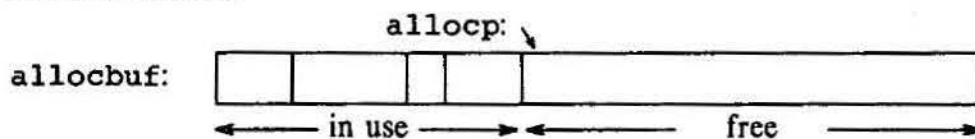
C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays, and address arithmetic is one of the strengths of the language. Let us illustrate by writing a rudimentary storage allocator. There are two routines. The first, *alloc*(*n*), returns a pointer *p* to *n* consecutive character positions, which can be used by the caller of *alloc* for storing characters. The second, *afree*(*p*), releases the storage thus acquired so it can be re-used later. The routines are “rudimentary” because the calls to *afree* must be made in the opposite order to the calls made on *alloc*. That is, the storage

managed by `alloc` and `afree` is a stack, or last-in, first-out list. The standard library provides analogous functions called `malloc` and `free` that have no such restrictions; in Section 8.7 we will show how they can be implemented.

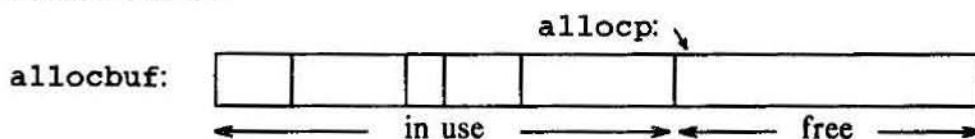
The easiest implementation is to have `alloc` hand out pieces of a large character array that we will call `allocbuf`. This array is private to `alloc` and `afree`. Since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared `static` in the source file containing `alloc` and `afree`, and thus be invisible outside it. In practical implementations, the array may well not even have a name; it might instead be obtained by calling `malloc` or by asking the operating system for a pointer to some unnamed block of storage.

The other information needed is how much of `allocbuf` has been used. We use a pointer, called `allocp`, that points to the next free element. When `alloc` is asked for `n` characters, it checks to see if there is enough room left in `allocbuf`. If so, `alloc` returns the current value of `allocp` (i.e., the beginning of the free block), then increments it by `n` to point to the next free area. If there is no room, `alloc` returns zero. `afree(p)` merely sets `allocp` to `p` if `p` is inside `allocbuf`.

before call to `alloc`:



after call to `alloc`:



```
#define ALLOCSIZE 10000 /* size of available space */

static char allocbuf[ALLOCSIZE]; /* storage for alloc */
static char *allocp = allocbuf; /* next free position */

char *alloc(int n) /* return pointer to n characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
        allocp += n;
        return allocp - n; /* old p */
    } else /* not enough room */
        return 0;
}
```



```
void afree(char *p) /* free storage pointed to by p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the addresses of previously defined data of appropriate type. The declaration

```
static char *allocp = allocbuf;
```

defines `allocp` to be a character pointer and initializes it to point to the beginning of `allocbuf`, which is the next free position when the program starts. This could have also been written

```
static char *allocp = &allocbuf[0];
```

since the array name *is* the address of the zeroth element.

The test

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
```

checks if there's enough room to satisfy a request for `n` characters. If there is, the new value of `allocp` would be at most one beyond the end of `allocbuf`. If the request can be satisfied, `alloc` returns a pointer to the beginning of a block of characters (notice the declaration of the function itself). If not, `alloc` must return some signal that no space is left. C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event, in this case, no space.

Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero. The symbolic constant `NULL` is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. `NULL` is defined in `<stdio.h>`. We will use `NULL` henceforth.

Tests like

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
```

and

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

show several important facets of pointer arithmetic. First, pointers may be compared under certain circumstances. If `p` and `q` point to members of the same array, then relations like `==`, `!=`, `<`, `>=`, etc., work properly. For example,

```
p < q
```

is true if `p` points to an earlier member of the array than `q` does. Any pointer can be meaningfully compared for equality or inequality with zero. But the behavior is undefined for arithmetic or comparisons with pointers that do not

point to members of the same array. (There is one exception: the address of the first element past the end of an array can be used in pointer arithmetic.)

Second, we have already observed that a pointer and an integer may be added or subtracted. The construction

$$p + n$$

means the address of the n -th object beyond the one p currently points to. This is true regardless of the kind of object p points to; n is scaled according to the size of the objects p points to, which is determined by the declaration of p . If an `int` is four bytes, for example, the `int` will be scaled by four.

Pointer subtraction is also valid: if p and q point to elements of the same array, and $p < q$, then $q - p + 1$ is the number of elements from p to q inclusive. This fact can be used to write yet another version of `strlen`:

```
/* strlen: return length of string s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

In its declaration, p is initialized to s , that is, to point to the first character of the string. In the `while` loop, each character in turn is examined until the `'\0'` at the end is seen. Because p points to characters, $p++$ advances p to the next character each time, and $p - s$ gives the number of characters advanced over, that is, the string length. (The number of characters in the string could be too large to store in an `int`. The header `<stddef.h>` defines a type `ptrdiff_t` that is large enough to hold the signed difference of two pointer values. If we were being very cautious, however, we would use `size_t` for the return type of `strlen`, to match the standard library version. `size_t` is the unsigned integer type returned by the `sizeof` operator.)

Pointer arithmetic is consistent: if we had been dealing with `float`s, which occupy more storage than `char`s, and if p were a pointer to `float`, $p++$ would advance to the next `float`. Thus we could write another version of `alloc` that maintains `float`s instead of `char`s, merely by changing `char` to `float` throughout `alloc` and `afree`. All the pointer manipulations automatically take into account the size of the object pointed to.

The valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero. All other pointer arithmetic is illegal. It is not legal to add two pointers, or to multiply or divide or shift or mask them, or to add `float` or `double` to them, or even, except for `void *`, to assign a pointer of one type to a pointer of another type without a cast.

5.5 Character Pointers and Functions

A *string constant*, written as

```
"I am a string"
```

is an array of characters. In the internal representation, the array is terminated with the null character `'\0'` so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as arguments to functions, as in

```
printf("hello, world\n");
```

When a character string like this appears in a program, access to it is through a character pointer; `printf` receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element.

String constants need not be function arguments. If `pmessage` is declared as

```
char *pmessage;
```

then the statement

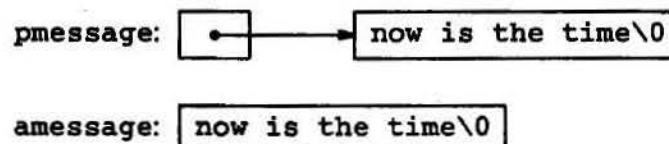
```
pmessage = "now is the time";
```

assigns to `pmessage` a pointer to the character array. This is *not* a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

There is an important difference between these definitions:

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */
```

`amessage` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amessage` will always refer to the same storage. On the other hand, `pmessage` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.



We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library. The first function is `strcpy(s,t)`, which copies the string `t` to the string `s`. It would be nice just to say `s=t` but this copies the pointer, not the characters. To copy the

characters, we need a loop. The array version is first:

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

For contrast, here is a version of `strcpy` with pointers:

```
/* strcpy: copy t to s; pointer version 1 */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Because arguments are passed by value, `strcpy` can use the parameters `s` and `t` in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the `'\0'` that terminates `t` has been copied to `s`.

In practice, `strcpy` would not be written as we showed it above. Experienced C programmers would prefer

```
/* strcpy: copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

This moves the increment of `s` and `t` into the test part of the loop. The value of `*t++` is the character that `t` pointed to before `t` was incremented; the postfix `++` doesn't change `t` until after this character has been fetched. In the same way, the character is stored into the old `s` position before `s` is incremented. This character is also the value that is compared against `'\0'` to control the loop. The net effect is that characters are copied from `t` to `s`, up to and including the terminating `'\0'`.

As the final abbreviation, observe that a comparison against `'\0'` is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

```

/* strcpy: copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}

```

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, because you will see it frequently in C programs.

The `strcpy` in the standard library (`<string.h>`) returns the target string as its function value.

The second routine that we will examine is `strcmp(s,t)`, which compares the character strings `s` and `t`, and returns negative, zero or positive if `s` is lexicographically less than, equal to, or greater than `t`. The value is obtained by subtracting the characters at the first position where `s` and `t` disagree.

```

/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}

```

The pointer version of `strcmp`:

```

/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}

```

Since `++` and `--` are either prefix or postfix operators, other combinations of `*` and `++` and `--` occur, although less frequently. For example,

```
*--p
```

decrements `p` before fetching the character that `p` points to. In fact, the pair of expressions

```

*p++ = val;    /* push val onto stack */
val = *--p;    /* pop top of stack into val */

```

are the standard idioms for pushing and popping a stack; see Section 4.3.

The header `<string.h>` contains declarations for the functions mentioned

in this section, plus a variety of other string-handling functions from the standard library.

Exercise 5-3. Write a pointer version of the function `strcat` that we showed in Chapter 2: `strcat(s,t)` copies the string `t` to the end of `s`. □

Exercise 5-4. Write the function `strend(s,t)`, which returns 1 if the string `t` occurs at the end of the string `s`, and zero otherwise. □

Exercise 5-5. Write versions of the library functions `strncpy`, `strncat`, and `strncmp`, which operate on at most the first `n` characters of their argument strings. For example, `strncpy(s,t,n)` copies at most `n` characters of `t` to `s`. Full descriptions are in Appendix B. □

Exercise 5-6. Rewrite appropriate programs from earlier chapters and exercises with pointers instead of array indexing. Good possibilities include `getline` (Chapters 1 and 4), `atoi`, `itoa`, and their variants (Chapters 2, 3, and 4), `reverse` (Chapter 3), and `strindex` and `getop` (Chapter 4). □

5.6 Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, they can be stored in arrays just as other variables can. Let us illustrate by writing a program that will sort a set of text lines into alphabetic order, a stripped-down version of the UNIX program `sort`.

In Chapter 3 we presented a Shell sort function that would sort an array of integers, and in Chapter 4 we improved on it with a quicksort. The same algorithms will work, except that now we have to deal with lines of text, which are of different lengths, and which, unlike integers, can't be compared or moved in a single operation. We need a data representation that will cope efficiently and conveniently with variable-length text lines.

This is where the array of pointers enters. If the lines to be sorted are stored end-to-end in one long character array, then each line can be accessed by a pointer to its first character. The pointers themselves can be stored in an array. Two lines can be compared by passing their pointers to `strcmp`. When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.



This eliminates the twin problems of complicated storage management and high overhead that would go with moving the lines themselves.

The sorting process has three steps:

*read all the lines of input
sort them
print them in order*

As usual, it's best to divide the program into functions that match this natural division, with the main routine controlling the other functions. Let us defer the sorting step for a moment, and concentrate on the data structure and the input and output.

The input routine has to collect and save the characters of each line, and build an array of pointers to the lines. It will also have to count the number of input lines, since that information is needed for sorting and printing. Since the input function can only cope with a finite number of input lines, it can return some illegal line count like -1 if too much input is presented.

The output routine only has to print the lines in the order in which they appear in the array of pointers.

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

#define MAXLINES 5000      /* max #lines to be sorted */

char *lineptr[MAXLINES];  /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(char *lineptr[], int left, int right);

/* sort input lines */
main()
{
    int nlines;      /* number of input lines read */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}
```

```

#define MAXLEN 1000 /* max length of any input line */
int getline(char *, int);
char *alloc(int);

/* readlines: read input lines */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* delete newline */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}

```

The function `getline` is from Section 1.9.

The main new thing is the declaration for `lineptr`:

```
char *lineptr[MAXLINES]
```

says that `lineptr` is an array of `MAXLINES` elements, each element of which is a pointer to a `char`. That is, `lineptr[i]` is a character pointer, and `*lineptr[i]` is the character it points to, the first character of the *i*-th saved text line.

Since `lineptr` is itself the name of an array, it can be treated as a pointer in the same manner as in our earlier examples, and `writelines` can be written instead as

```

/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}

```


Initially `*lineptr` points to the first line; each increment advances it to the next line pointer while `nlines` is counted down.

With input and output under control, we can proceed to sorting. The quick-sort from Chapter 4 needs minor changes: the declarations have to be modified, and the comparison operation must be done by calling `strcmp`. The algorithm remains the same, which gives us some confidence that it will still work.

```
/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right) /* do nothing if array contains */
        return;       /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Similarly, the swap routine needs only trivial changes:

```
/* swap:  interchange v[i] and v[j] */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Since any individual element of `v` (alias `lineptr`) is a character pointer, `temp` must be also, so one can be copied to the other.

Exercise 5-7. Rewrite `readlines` to store lines in an array supplied by `main`, rather than calling `alloc` to maintain storage. How much faster is the program? □

5.7 Multi-dimensional Arrays

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers. In this section, we will show some of their properties.

Consider the problem of date conversion, from day of the month to day of the year and vice versa. For example, March 1 is the 60th day of a non-leap year, and the 61st day of a leap year. Let us define two functions to do the conversions: `day_of_year` converts the month and day into the day of the year, and `month_day` converts the day of the year into the month and day. Since this latter function computes two values, the month and day arguments will be pointers:

```
month_day(1988, 60, &m, &d)
```

sets `m` to 2 and `d` to 29 (February 29th).

These functions both need the same information, a table of the number of days in each month ("thirty days hath September ..."). Since the number of days per month differs for leap years and non-leap years, it's easier to separate them into two rows of a two-dimensional array than to keep track of what happens to February during computation. The array and the functions for performing the transformations are as follows:

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: set day of year from month & day */
int day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day: set month, day from day of year */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

Recall that the arithmetic value of a logical expression, such as the one for `leap`, is either zero (false) or one (true), so it can be used as a subscript of the array `daytab`.

The array `daytab` has to be external to both `day_of_year` and

month_day, so they can both use it. We made it char to illustrate a legitimate use of char for storing small non-character integers.

daytab is the first two-dimensional array we have dealt with. In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array. Hence subscripts are written as

```
daytab[i][j]    /* [row][col] */
```

rather than

```
daytab[i,j]    /* WRONG */
```

Other than this notational distinction, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order.

An array is initialized by a list of initializers in braces; each row of a two-dimensional array is initialized by a corresponding sub-list. We started the array daytab with a column of zero so that month numbers can run from the natural 1 to 12 instead of 0 to 11. Since space is not at a premium here, this is clearer than adjusting the indices.

If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of columns; the number of rows is irrelevant, since what is passed is, as before, a pointer to an array of rows, where each row is an array of 13 ints. In this particular case, it is a pointer to objects that are arrays of 13 ints. Thus if the array daytab is to be passed to a function f, the declaration of f would be

```
f(int daytab[2][13]) { ... }
```

It could also be

```
f(int daytab[][13]) { ... }
```

since the number of rows is irrelevant, or it could be

```
f(int (*daytab)[13]) { ... }
```

which says that the parameter is a pointer to an array of 13 integers. The parentheses are necessary since brackets [] have higher precedence than *. Without parentheses, the declaration

```
int *daytab[13]
```

is an array of 13 pointers to integers. More generally, only the first dimension (subscript) of an array is free; all the others have to be specified.

Section 5.12 has a further discussion of complicated declarations.

Exercise 5-8. There is no error checking in day_of_year or month_day. Remedy this defect. □

5.8 Initialization of Pointer Arrays

Consider the problem of writing a function `month_name(n)`, which returns a pointer to a character string containing the name of the n -th month. This is an ideal application for an internal static array. `month_name` contains a private array of character strings, and returns a pointer to the proper one when called. This section shows how that array of names is initialized.

The syntax is similar to previous initializations:

```
/* month_name: return name of n-th month */
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

The declaration of `name`, which is an array of character pointers, is the same as `lineptr` in the sorting example. The initializer is a list of character strings; each is assigned to the corresponding position in the array. The characters of the i -th string are placed somewhere, and a pointer to them is stored in `name[i]`. Since the size of the array `name` is not specified, the compiler counts the initializers and fills in the correct number.

5.9 Pointers vs. Multi-dimensional Arrays

Newcomers to C are sometimes confused about the difference between a two-dimensional array and an array of pointers, such as `name` in the example above. Given the definitions

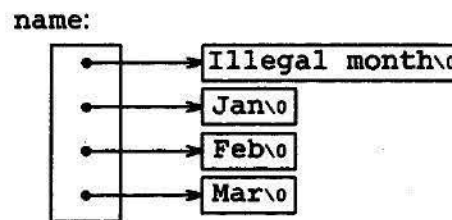
```
int a[10][20];
int *b[10];
```

then `a[3][4]` and `b[3][4]` are both syntactically legal references to a single `int`. But `a` is a true two-dimensional array: 200 `int`-sized locations have been set aside, and the conventional rectangular subscript calculation $20 \times \text{row} + \text{col}$ is used to find the element `a[row][col]`. For `b`, however, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either statically or with code. Assuming that each element of `b` does point to a twenty-element array, then there will be 200 `ints` set aside, plus ten cells for the pointers. The important advantage of the pointer array is that the rows of the array may be of different lengths. That is, each element of `b` need not

point to a twenty-element vector; some may point to two elements, some to fifty, and some to none at all.

Although we have phrased this discussion in terms of integers, by far the most frequent use of arrays of pointers is to store character strings of diverse lengths, as in the function `month_name`. Compare the declaration and picture for an array of pointers:

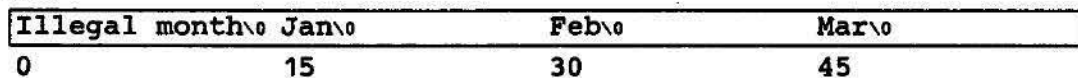
```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
```



with those for a two-dimensional array:

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

aname:



Exercise 5-9. Rewrite the routines `day_of_year` and `month_day` with pointers instead of indexing. □

5.10 Command-line Arguments

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When `main` is called, it is called with two arguments. The first (conventionally called `argc`, for argument count) is the number of command-line arguments the program was invoked with; the second (`argv`, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

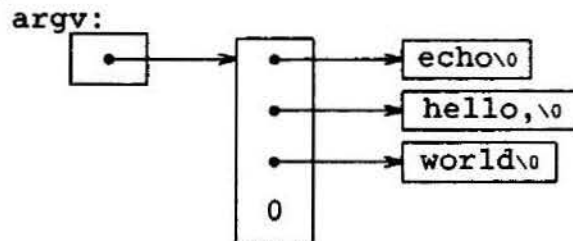
The simplest illustration is the program `echo`, which echoes its command-line arguments on a single line, separated by blanks. That is, the command

```
echo hello, world
```

prints the output

```
hello, world
```

By convention, `argv[0]` is the name by which the program was invoked, so `argc` is at least 1. If `argc` is 1, there are no command-line arguments after the program name. In the example above, `argc` is 3, and `argv[0]`, `argv[1]`, and `argv[2]` are "echo", "hello,", and "world" respectively. The first optional argument is `argv[1]` and the last is `argv[argc-1]`; additionally, the standard requires that `argv[argc]` be a null pointer.



The first version of `echo` treats `argv` as an array of character pointers:

```
#include <stdio.h>

/* echo command-line arguments; 1st version */
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since `argv` is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. This next variation is based on incrementing `argv`, which is a pointer to pointer to `char`, while `argc` is counted down:

```
#include <stdio.h>

/* echo command-line arguments; 2nd version */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", ++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since `argv` is a pointer to the beginning of the array of argument strings, incrementing it by 1 (`++argv`) makes it point at the original `argv[1]` instead of `argv[0]`. Each successive increment moves it along to the next argument; `*argv` is then the pointer to that argument. At the same time, `argc` is decremented; when it becomes zero, there are no arguments left to print.

Alternatively, we could write the `printf` statement as

```
printf((argc > 1) ? "%s " : "%s", ++argv);
```

This shows that the format argument of `printf` can be an expression too.

As a second example, let us make some enhancements to the pattern-finding program from Section 4.1. If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the UNIX program `grep`, let us change the program so the pattern to be matched is specified by the first argument on the command line.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

The standard library function `strstr(s,t)` returns a pointer to the first occurrence of the string `t` in the string `s`, or `NULL` if there is none. It is declared in `<string.h>`.

The model can now be elaborated to illustrate further pointer constructions. Suppose we want to allow two optional arguments. One says “print all lines *except* those that match the pattern;” the second says “precede each printed line by its line number.”

A common convention for C programs on UNIX systems is that an argument that begins with a minus sign introduces an optional flag or parameter. If we choose `-x` (for “except”) to signal the inversion, and `-n` (“number”) to request line numbering, then the command

```
find -x -n pattern
```

will print each line that doesn’t match the pattern, preceded by its line number.

Optional arguments should be permitted in any order, and the rest of the program should be independent of the number of arguments that were present. Furthermore, it is convenient for users if option arguments can be combined, as

in

```
find -nx pattern
```

Here is the program:

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
                case 'x':
                    except = 1;
                    break;
                case 'n':
                    number = 1;
                    break;
                default:
                    printf("find: illegal option %c\n", c);
                    argc = 0;
                    found = -1;
                    break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

`argc` is decremented and `argv` is incremented before each optional argument. At the end of the loop, if there are no errors, `argc` tells how many arguments remain unprocessed and `argv` points to the first of these. Thus `argc`

should be 1 and `*argv` should point at the pattern. Notice that `***argv` is a pointer to an argument string, so `(***argv)[0]` is its first character. (An alternate valid form would be `****argv`.) Because `[]` binds tighter than `*` and `++`, the parentheses are necessary; without them the expression would be taken as `++(argv[0])`. In fact, that is what we used in the inner loop, where the task is to walk along a specific argument string. In the inner loop, the expression `***argv[0]` increments the pointer `argv[0]`!

It is rare that one uses pointer expressions more complicated than these; in such cases, breaking them into two or three steps will be more intuitive.

Exercise 5-10. Write the program `expr`, which evaluates a reverse Polish expression from the command line, where each operator or operand is a separate argument. For example,

```
expr 2 3 4 + *
```

evaluates $2 \times (3+4)$. \square

Exercise 5-11. Modify the programs `entab` and `detab` (written as exercises in Chapter 1) to accept a list of tab stops as arguments. Use the default tab settings if there are no arguments. \square

Exercise 5-12. Extend `entab` and `detab` to accept the shorthand

```
entab -m +n
```

to mean tab stops every n columns, starting at column m . Choose convenient (for the user) default behavior. \square

Exercise 5-13. Write the program `tail`, which prints the last n lines of its input. By default, n is 10, let us say, but it can be changed by an optional argument, so that

```
tail -n
```

prints the last n lines. The program should behave rationally no matter how unreasonable the input or the value of n . Write the program so it makes the best use of available storage; lines should be stored as in the sorting program of Section 5.6, not in a two-dimensional array of fixed size. \square

5.11 Pointers to Functions

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on. We will illustrate this by modifying the sorting procedure written earlier in this chapter so that if the optional argument `-n` is given, it will sort the input lines numerically instead of lexicographically.

A sort often consists of three parts—a comparison that determines the

ordering of any pair of objects, an exchange that reverses their order, and a sorting algorithm that makes comparisons and exchanges until the objects are in order. The sorting algorithm is independent of the comparison and exchange operations, so by passing different comparison and exchange functions to it, we can arrange to sort by different criteria. This is the approach taken in our new sort.

Lexicographic comparison of two lines is done by `strcmp`, as before; we will also need a routine `numcmp` that compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp` does. These functions are declared ahead of `main` and a pointer to the appropriate one is passed to `qsort`. We have skimmed on error processing for arguments, so as to concentrate on the main issues.

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

#define MAXLINES 5000      /* max #lines to be sorted */
char *lineptr[MAXLINES]; /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
int numcmp(char *, char *);

/* sort input lines */
main(int argc, char *argv[])
{
    int nlines;          /* number of input lines read */
    int numeric = 0;      /* 1 if numeric sort */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void **) lineptr, 0, nlines-1,
              (int (*)(void*,void*)) (numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}
```

In the call to `qsort`, `strcmp` and `numcmp` are addresses of functions. Since they are known to be functions, the `&` operator is not necessary, in the same way that it is not needed before an array name.

We have written `qsort` so it can process any data type, not just character

strings. As indicated by the function prototype, `qsort` expects an array of pointers, two integers, and a function with two pointer arguments. The generic pointer type `void *` is used for the pointer arguments. Any pointer can be cast to `void *` and back again without loss of information, so we can call `qsort` by casting arguments to `void *`. The elaborate cast of the function argument casts the arguments of the comparison function. These will generally have no effect on actual representation, but assure the compiler that all is well.

```
/* qsort: sort v[left]...v[right] into increasing order */
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[], int, int);

    if (left >= right) /* do nothing if array contains */
        return;       /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
```

The declarations should be studied with some care. The fourth parameter of `qsort` is

```
int (*comp)(void *, void *)
```

which says that `comp` is a pointer to a function that has two `void *` arguments and returns an `int`.

The use of `comp` in the line

```
if ((*comp)(v[i], v[left]) < 0)
```

is consistent with the declaration: `comp` is a pointer to a function, `*comp` is the function, and

```
(*comp)(v[i], v[left])
```

is the call to it. The parentheses are needed so the components are correctly associated; without them,

```
int *comp(void *, void *) /* WRONG */
```

says that `comp` is a function returning a pointer to an `int`, which is very different.

We have already shown `strcmp`, which compares two strings. Here is `numcmp`, which compares two strings on a leading numeric value, computed by

calling `atof`:

```
#include <stdlib.h>

/* numcmp: compare s1 and s2 numerically */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

The `swap` function, which exchanges two pointers, is identical to what we presented earlier in the chapter, except that the declarations are changed to `void *`.

```
void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

A variety of other options can be added to the sorting program; some make challenging exercises.

Exercise 5-14. Modify the sort program to handle a `-r` flag, which indicates sorting in reverse (decreasing) order. Be sure that `-r` works with `-n`. □

Exercise 5-15. Add the option `-f` to fold upper and lower case together, so that case distinctions are not made during sorting; for example, `a` and `A` compare equal. □

Exercise 5-16. Add the `-d` (“directory order”) option, which makes comparisons only on letters, numbers and blanks. Make sure it works in conjunction with `-f`. □

Exercise 5-17. Add a field-handling capability, so sorting may be done on fields within lines, each field sorted according to an independent set of options. (The index for this book was sorted with `-df` for the index category and `-n` for the page numbers.) □

5.12 Complicated Declarations

C is sometimes castigated for the syntax of its declarations, particularly ones that involve pointers to functions. The syntax is an attempt to make the declaration and the use agree; it works well for simple cases, but it can be confusing for the harder ones, because declarations cannot be read left to right, and because parentheses are over-used. The difference between

```
int *f();      /* f: function returning pointer to int */
```

and

```
int (*pf)();   /* pf: pointer to function returning int */
```

illustrates the problem: `*` is a prefix operator and it has lower precedence than `()`, so parentheses are necessary to force the proper association.

Although truly complicated declarations rarely arise in practice, it is important to know how to understand them, and, if necessary, how to create them. One good way to synthesize declarations is in small steps with `typedef`, which is discussed in Section 6.7. As an alternative, in this section we will present a pair of programs that convert from valid C to a word description and back again. The word description reads left to right.

The first, `dcl`, is the more complex. It converts a C declaration into a word description, as in these examples:

```
char **argv
    argv: pointer to pointer to char
int (*daytab)[13]
    daytab: pointer to array[13] of int
int *daytab[13]
    daytab: array[13] of pointer to int
void *comp()
    comp: function returning pointer to void
void (*comp)()
    comp: pointer to function returning void
char ((*x())[5])()
    x: function returning pointer to array[] of
    pointer to function returning char
char ((*x[3])())[5]
    x: array[3] of pointer to function returning
    pointer to array[5] of char
```

`dcl` is based on the grammar that specifies a declarator, which is spelled out precisely in Appendix A, Section 8.5; this is a simplified form:

```
dcl:      optional *'s direct-dcl
direct-dcl: name
            (dcl)
            direct-dcl()
            direct-dcl[optional size]
```

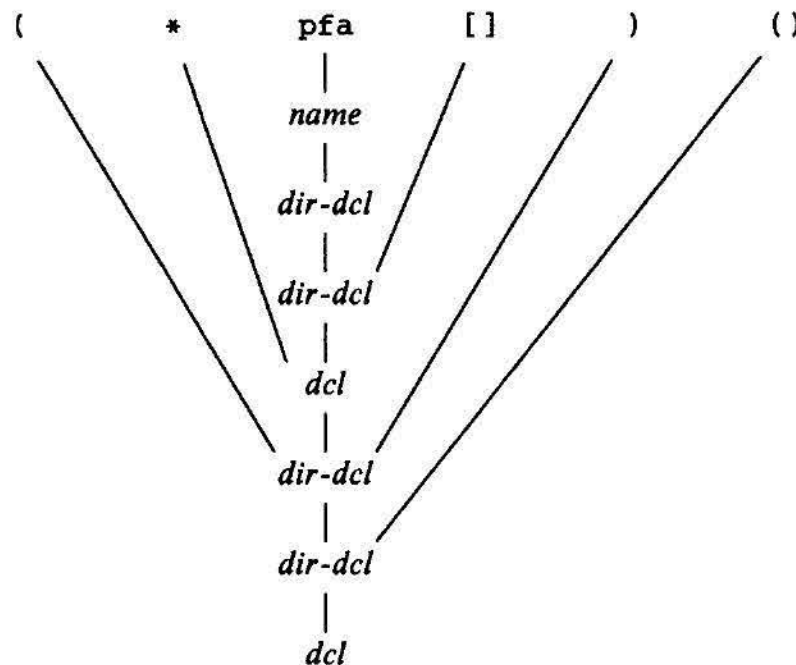
In words, a *dcl* is a *direct-dcl*, perhaps preceded by **'s*. A *direct-dcl* is a

name, or a parenthesized *dcl*, or a *direct-dcl* followed by parentheses, or a *direct-dcl* followed by brackets with an optional size.

This grammar can be used to parse declarations. For instance, consider this declarator:

```
(*pfa[])( )
```

pfa will be identified as a *name* and thus as a *direct-dcl*. Then *pfa[]* is also a *direct-dcl*. Then **pfa[]* is recognized as a *dcl*, so *(*pfa[])* is a *direct-dcl*. Then *(*pfa[])()* is a *direct-dcl* and thus a *dcl*. We can also illustrate the parse with a parse tree like this (where *direct-dcl* has been abbreviated to *dir-dcl*):



The heart of the *dcl* program is a pair of functions, *dcl* and *dirdcl*, that parse a declaration according to this grammar. Because the grammar is recursively defined, the functions call each other recursively as they recognize pieces of a declaration; the program is called a recursive-descent parser.

```

/* dcl:  parse a declarator */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*'; ) /* count *'s */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " pointer to");
}

```

```

/* dirdcl: parse a direct declarator */
void dirdcl(void)
{
    int type;

    if (tokentype == '(') {          /* ( dcl ) */
        dcl();
        if (tokentype != ')')
            printf("error: missing )\n");
    } else if (tokentype == NAME)    /* variable name */
        strcpy(name, token);
    else
        printf("error: expected name or (dcl)\n");
    while ((type=gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, " function returning");
        else {
            strcat(out, " array");
            strcat(out, token);
            strcat(out, " of");
        }
}

```

Since the programs are intended to be illustrative, not bullet-proof, there are significant restrictions on `dcl`. It can only handle a simple data type like `char` or `int`. It does not handle argument types in functions, or qualifiers like `const`. Spurious blanks confuse it. It doesn't do much error recovery, so invalid declarations will also confuse it. These improvements are left as exercises.

Here are the global variables and the main routine:

```

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

#define MAXTOKEN 100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int gettoken(void);
int tokentype;          /* type of last token */
char token[MAXTOKEN];   /* last token string */
char name[MAXTOKEN];    /* identifier name */
char datatype[MAXTOKEN]; /* data type = char, int, etc. */
char out[1000];         /* output string */

```

```

main() /* convert declaration to words */
{
    while (gettoken() != EOF) { /* 1st token on line */
        strcpy(datatype, token); /* is the datatype */
        out[0] = '\0';
        dcl(); /* parse rest of line */
        if (tokentype != '\n')
            printf("syntax error\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}

```

The function `gettoken` skips blanks and tabs, then finds the next token in the input; a "token" is a name, a pair of parentheses, a pair of brackets perhaps including a number, or any other single character.

```

int gettoken(void) /* return next token */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;

    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy(token, "()");
            return tokentype = PARENS;
        } else {
            ungetch(c);
            return tokentype = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
        ungetch(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}

```

`getch` and `ungetch` were discussed in Chapter 4.

Going in the other direction is easier, especially if we do not worry about generating redundant parentheses. The program `undcl` converts a word

description like “*x* is a function returning a pointer to an array of pointers to functions returning *char*,” which we will express as

```
x () * [] * () char
```

to

```
char ((*x())[ ] )()
```

The abbreviated input syntax lets us reuse the `gettoken` function. `undcl` also uses the same external variables as `dcl` does.

```
/* undcl: convert word description to declaration */
main()
{
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, "(*%s)", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("invalid input at %s\n", token);
        printf("%s\n", out);
    }
    return 0;
}
```

Exercise 5-18. Make `dcl` recover from input errors. □

Exercise 5-19. Modify `undcl` so that it does not add redundant parentheses to declarations. □

Exercise 5-20. Expand `dcl` to handle declarations with function argument types, qualifiers like `const`, and so on. □

CHAPTER 7: **Input and Output**

Input and output facilities are not part of the C language itself, so we have not emphasized them in our presentation thus far. Nonetheless, programs interact with their environment in much more complicated ways than those we have shown before. In this chapter we will describe the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs. We will concentrate on input and output.

The ANSI standard defines these library functions precisely, so that they can exist in compatible form on any system where C exists. Programs that confine their system interactions to facilities provided by the standard library can be moved from one system to another without change.

The properties of library functions are specified in more than a dozen headers; we have already seen several of these, including `<stdio.h>`, `<string.h>`, and `<ctype.h>`. We will not present the entire library here, since we are more interested in writing C programs that use it. The library is described in detail in Appendix B.

7.1 Standard Input and Output

As we said in Chapter 1, the library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character. If the system doesn't operate that way, the library does whatever is necessary to make it appear as if it does. For instance, the library might convert carriage return and linefeed to newline on input and back again on output.

The simplest input mechanism is to read one character at a time from the *standard input*, normally the keyboard, with `getchar`:

```
int getchar(void)
```

`getchar` returns the next input character each time it is called, or EOF when it encounters end of file. The symbolic constant EOF is defined in `<stdio.h>`.

The value is typically `-1`, but tests should be written in terms of `EOF` so as to be independent of the specific value.

In many environments, a file may be substituted for the keyboard by using the `<` convention for input redirection: if a program `prog` uses `getchar`, then the command line

```
prog <infile
```

causes `prog` to read characters from `infile` instead. The switching of the input is done in such a way that `prog` itself is oblivious to the change; in particular, the string `<infile` is not included in the command-line arguments in `argv`. Input switching is also invisible if the input comes from another program via a pipe mechanism: on some systems, the command line

```
otherprog | prog
```

runs the two programs `otherprog` and `prog`, and pipes the standard output of `otherprog` into the standard input for `prog`.

The function

```
int putchar(int)
```

is used for output: `putchar(c)` puts the character `c` on the *standard output*, which is by default the screen. `putchar` returns the character written, or `EOF` if an error occurs. Again, output can usually be directed to a file with `>filename`: if `prog` uses `putchar`,

```
prog >outfile
```

will write the standard output to `outfile` instead. If pipes are supported,

```
prog | anotherprog
```

puts the standard output of `prog` into the standard input of `anotherprog`.

Output produced by `printf` also finds its way to the standard output. Calls to `putchar` and `printf` may be interleaved—output appears in the order in which the calls were made.

Each source file that refers to an input/output library function must contain the line

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

before the first reference. When the name is bracketed by `<` and `>` a search is made for the header in a standard set of places (for example, on UNIX systems, typically in the directory `/usr/include`).

Many programs read only one input stream and write only one output stream; for such programs, input and output with `getchar`, `putchar`, and `printf` may be entirely adequate, and is certainly enough to get started. This is particularly true if redirection is used to connect the output of one program to the input of the next. For example, consider the program `lower`, which converts its input to lower case:

```

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

main() /* lower: convert input to lower case */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}

```

The function `tolower` is defined in `<ctype.h>`; it converts an upper case letter to lower case, and returns other characters untouched. As we mentioned earlier, “functions” like `getchar` and `putchar` in `<stdio.h>` and `tolower` in `<ctype.h>` are often macros, thus avoiding the overhead of a function call per character. We will show how this is done in Section 8.5. Regardless of how the `<ctype.h>` functions are implemented on a given machine, programs that use them are shielded from knowledge of the character set.

Exercise 7-1. Write a program that converts upper case to lower or lower case to upper, depending on the name it is invoked with, as found in `argv[0]`. □

7.2 Formatted Output—Printf

The output function `printf` translates internal values to characters. We have used `printf` informally in previous chapters. The description here covers most typical uses but is not complete; for the full story, see Appendix B.

```
int printf(char *format, arg1, arg2, ...)
```

`printf` converts, formats, and prints its arguments on the standard output under control of the `format`. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to `printf`. Each conversion specification begins with a `%` and ends with a conversion character. Between the `%` and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.

- An `h` if the integer is to be printed as a short, or `l` (letter ell) if as a long.

Conversion characters are shown in Table 7-1. If the character after the `%` is not a conversion specification, the behavior is undefined.

TABLE 7-1. BASIC PRINTF CONVERSIONS

CHARACTER	ARGUMENT TYPE; PRINTED AS
<code>d, i</code>	<code>int</code> ; decimal number.
<code>o</code>	<code>int</code> ; unsigned octal number (without a leading zero).
<code>x, X</code>	<code>int</code> ; unsigned hexadecimal number (without a leading <code>0x</code> or <code>0X</code>), using <code>abcdef</code> or <code>ABCDEF</code> for 10, ..., 15.
<code>u</code>	<code>int</code> ; unsigned decimal number.
<code>c</code>	<code>int</code> ; single character.
<code>s</code>	<code>char *</code> ; print characters from the string until a <code>'\0'</code> or the number of characters given by the precision.
<code>f</code>	<code>double</code> ; <code>[-]m.dddddd</code> , where the number of <code>d</code> 's is given by the precision (default 6).
<code>e, E</code>	<code>double</code> ; <code>[-]m.ddddde±xx</code> or <code>[-]m.dddddE±xx</code> , where the number of <code>d</code> 's is given by the precision (default 6).
<code>g, G</code>	<code>double</code> ; use <code>%e</code> or <code>%E</code> if the exponent is less than <code>-4</code> or greater than or equal to the precision; otherwise use <code>%f</code> . Trailing zeros and a trailing decimal point are not printed.
<code>p</code>	<code>void *</code> ; pointer (implementation-dependent representation).
<code>%</code>	no argument is converted; print a <code>%</code> .

A width or precision may be specified as `*`, in which case the value is computed by converting the next argument (which must be an `int`). For example, to print at most `max` characters from a string `s`,

```
printf("%.*s", max, s);
```

Most of the format conversions have been illustrated in earlier chapters. One exception is precision as it relates to strings. The following table shows the effect of a variety of specifications in printing "hello, world" (12 characters). We have put colons around each field so you can see its extent.

```
:%s:           :hello, world:
:%10s:         :hello, world:
:%.10s:         :hello, wor:
:%-10s:         :hello, world:
:%.15s:         :hello, world:
:%-15s:         :hello, world :
:%15.10s:       :   hello, wor:
:%-15.10s:      :hello, wor   :
```

A warning: `printf` uses its first argument to decide how many arguments

follow and what their types are. It will get confused, and you will get wrong answers, if there are not enough arguments or if they are the wrong type. You should also be aware of the difference between these two calls:

```
printf(s);          /* FAILS if s contains % */
printf("%s", s);    /* SAFE */
```

The function `sprintf` does the same conversions as `printf` does, but stores the output in a string:

```
int sprintf(char *string, char *format, arg1, arg2, ...)
```

`sprintf` formats the arguments in `arg1`, `arg2`, etc., according to `format` as before, but places the result in `string` instead of on the standard output; `string` must be big enough to receive the result.

Exercise 7-2. Write a program that will print arbitrary input in a sensible way. As a minimum, it should print non-graphic characters in octal or hexadecimal according to local custom, and break long text lines. □

7.3 Variable-length Argument Lists

This section contains an implementation of a minimal version of `printf`, to show how to write a function that processes a variable-length argument list in a portable way. Since we are mainly interested in the argument processing, `minprintf` will process the format string and arguments but will call the real `printf` to do the format conversions.

The proper declaration for `printf` is

```
int printf(char *fmt, ...)
```

where the declaration `...` means that the number and types of these arguments may vary. The declaration `...` can only appear at the end of an argument list. Our `minprintf` is declared as

```
void minprintf(char *fmt, ...)
```

since we will not return the character count that `printf` does.

The tricky bit is how `minprintf` walks along the argument list when the list doesn't even have a name. The standard header `<stdarg.h>` contains a set of macro definitions that define how to step through an argument list. The implementation of this header will vary from machine to machine, but the interface it presents is uniform.

The type `va_list` is used to declare a variable that will refer to each argument in turn; in `minprintf`, this variable is called `ap`, for "argument pointer." The macro `va_start` initializes `ap` to point to the first unnamed argument. It must be called once before `ap` is used. There must be at least one named argument; the final named argument is used by `va_start` to get started.

Each call of `va_arg` returns one argument and steps ap to the next; `va_arg` uses a type name to determine what type to return and how big a step to take. Finally, `va_end` does whatever cleanup is necessary. It must be called before the function returns.

These properties form the basis of our simplified `printf`:

```
#include <stdarg.h>

/* minprintf: minimal printf with variable argument list */
void minprintf(char *fmt, ...)
{
    va_list ap; /* points to each unnamed arg in turn */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt); /* make ap point to 1st unnamed arg */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
            case 'd':
                ival = va_arg(ap, int);
                printf("%d", ival);
                break;
            case 'f':
                dval = va_arg(ap, double);
                printf("%f", dval);
                break;
            case 's':
                for (sval = va_arg(ap, char *); *sval; sval++)
                    putchar(*sval);
                break;
            default:
                putchar(*p);
                break;
        }
    }
    va_end(ap); /* clean up when done */
}
```

Exercise 7-3. Revise `minprintf` to handle more of the other facilities of `printf`. □

7.4 Formatted Input—Scanf

The function `scanf` is the input analog of `printf`, providing many of the same conversion facilities in the opposite direction.

```
int scanf(char *format, ...)
```

`scanf` reads characters from the standard input, interprets them according to the specification in `format`, and stores the results through the remaining arguments. The format argument is described below; the other arguments, *each of which must be a pointer*, indicate where the corresponding converted input should be stored. As with `printf`, this section is a summary of the most useful features, not an exhaustive list.

`scanf` stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to `scanf` resumes searching immediately after the last character already converted.

There is also a function `sscanf` that reads from a string instead of the standard input:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

It scans the `string` according to the format in `format`, and stores the resulting values through `arg1`, `arg2`, etc. These arguments must be pointers.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l, or L indicating the width of the target, and a conversion character.

A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that `scanf` will read across line boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer, as required by the call-by-value

semantics of C. Conversion characters are shown in Table 7-2.

TABLE 7-2. BASIC SCANF CONVERSIONS

CHARACTER	INPUT DATA; ARGUMENT TYPE
d	decimal integer; <code>int *</code> .
i	integer; <code>int *</code> . The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	octal integer (with or without leading zero); <code>int *</code> .
u	unsigned decimal integer; unsigned <code>int *</code> .
x	hexadecimal integer (with or without leading 0x or 0X); <code>int *</code> .
c	characters; <code>char *</code> . The next input characters (default 1) are placed at the indicated spot. The normal skip over white space is suppressed; to read the next non-white space character, use <code>%1s</code> .
s	character string (not quoted); <code>char *</code> , pointing to an array of characters large enough for the string and a terminating <code>'\0'</code> that will be added.
e, f, g	floating-point number with optional sign, optional decimal point and optional exponent; <code>float *</code> .
%	literal %; no assignment is made.

The conversion characters `d`, `i`, `o`, `u`, and `x` may be preceded by `h` to indicate that a pointer to `short` rather than `int` appears in the argument list, or by `l` (letter ell) to indicate that a pointer to `long` appears in the argument list. Similarly, the conversion characters `e`, `f`, and `g` may be preceded by `l` to indicate that a pointer to `double` rather than `float` is in the argument list.

As a first example, the rudimentary calculator of Chapter 4 can be written with `scanf` to do the input conversion:

```
#include <stdio.h>

main() /* rudimentary calculator */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Suppose we want to read input lines that contain dates of the form

25 Dec 1988

The `scanf` statement is

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

No & is used with monthname, since an array name is a pointer.

Literal characters can appear in the `scanf` format string; they must match the same characters in the input. So we could read dates of the form `mm/dd/yy` with this `scanf` statement:

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

`scanf` ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with `sscanf`. For example, suppose we want to read lines that might contain a date in either of the forms above. Then we could write

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line);    /* 25 Dec 1988 form */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line);    /* mm/dd/yy form */
    else
        printf("invalid: %s\n", line); /* invalid form */
}
```

Calls to `scanf` can be mixed with calls to other input functions. The next call to any input function will begin by reading the first character not read by `scanf`.

A final warning: the arguments to `scanf` and `sscanf` *must* be pointers. By far the most common error is writing

```
scanf("%d", n);
```

instead of

```
scanf("%d", &n);
```

This error is not generally detected at compile time.

Exercise 7-4. Write a private version of `scanf` analogous to `minprintf` from the previous section. □

Exercise 7-5. Rewrite the postfix calculator of Chapter 4 to use `scanf` and/or `sscanf` to do the input and number conversion. □

7.5 File Access

The examples so far have all read the standard input and written the standard output, which are automatically defined for a program by the local operating system.

The next step is to write a program that accesses a file that is *not* already connected to the program. One program that illustrates the need for such operations is `cat`, which concatenates a set of named files onto the standard output. `cat` is used for printing files on the screen, and as a general-purpose input collector for programs that do not have the capability of accessing files by name. For example, the command

```
cat x.c y.c
```

prints the contents of the files `x.c` and `y.c` (and nothing else) on the standard output.

The question is how to arrange for the named files to be read—that is, how to connect the external names that a user thinks of to the statements that read the data.

The rules are simple. Before it can be read or written, a file has to be *opened* by the library function `fopen`. `fopen` takes an external name like `x.c` or `y.c`, does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns a pointer to be used in subsequent reads or writes of the file.

This pointer, called the *file pointer*, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred. Users don't need to know the details, because the definitions obtained from `<stdio.h>` include a structure declaration called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fp;  
FILE *fopen(char *name, char *mode);
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. Notice that `FILE` is a type name, like `int`, not a structure tag; it is defined with a `typedef`. (Details of how `fopen` can be implemented on the UNIX system are given in Section 8.5.)

The call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is a character string containing the name of the file. The second argument is the *mode*, also a character string, which indicates how one intends to use the file. Allowable modes include read ("`r`"), write ("`w`"), and append ("`a`"). Some systems distinguish between text and binary files; for the latter, a "`b`" must be appended to the mode string.

If a file that does not exist is opened for writing or appending, it is created if possible. Opening an existing file for writing causes the old contents to be discarded, while opening for appending preserves them. Trying to read a file that does not exist is an error, and there may be other causes of error as well, like trying to read a file when you don't have permission. If there is any error, `fopen` will return `NULL`. (The error can be identified more precisely; see the discussion of error-handling functions at the end of Section 1 in Appendix B.)

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it which file.

```
int getc(FILE *fp)
```

`getc` returns the next character from the stream referred to by `fp`; it returns `EOF` for end of file or error.

`putc` is an output function:

```
int putc(int c, FILE *fp)
```

`putc` writes the character `c` to the file `fp` and returns the character written, or `EOF` if an error occurs. Like `getchar` and `putchar`, `getc` and `putc` may be macros instead of functions.

When a C program is started, the operating system environment is responsible for opening three files and providing file pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`, and are declared in `<stdio.h>`. Normally `stdin` is connected to the keyboard and `stdout` and `stderr` are connected to the screen, but `stdin` and `stdout` may be redirected to files or pipes as described in Section 7.1.

`getchar` and `putchar` can be defined in terms of `getc`, `putc`, `stdin`, and `stdout` as follows:

```
#define getchar()    getc(stdin)
#define putchar(c)   putc((c), stdout)
```

For formatted input or output of files, the functions `fscanf` and `fprintf` may be used. These are identical to `scanf` and `printf`, except that the first argument is a file pointer that specifies the file to be read or written; the format string is the second argument.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

With these preliminaries out of the way, we are now in a position to write the program `cat` to concatenate files. The design is one that has been found convenient for many programs. If there are command-line arguments, they are interpreted as filenames, and processed in order. If there are no arguments, the standard input is processed.


```

#include <stdio.h>

/* cat: concatenate files, version 1 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);

    if (argc == 1) /* no args; copy standard input */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(++argv, "r")) == NULL) {
                printf("cat: can't open %s\n", *argv);
                return 1;
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
        return 0;
}

/* filecopy: copy file ifp to file ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}

```

The file pointers `stdin` and `stdout` are objects of type `FILE *`. They are constants, however, *not* variables, so it is not possible to assign to them.

The function

```
int fclose(FILE *fp)
```

is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since most operating systems have some limit on the number of files that a program may have open simultaneously, it's a good idea to free file pointers when they are no longer needed, as we did in `cat`. There is also another reason for `fclose` on an output file—it flushes the buffer in which `putc` is collecting output. `fclose` is called automatically for each open file when a program terminates normally. (You can close `stdin` and `stdout` if they are not needed. They can also be reassigned by the library function `freopen`.)

7.6 Error Handling—Stderr and Exit

The treatment of errors in `cat` is not ideal. The trouble is that if one of the files can't be accessed for some reason, the diagnostic is printed at the end of the concatenated output. That might be acceptable if the output is going to a screen, but not if it's going into a file or into another program via a pipeline.

To handle this situation better, a second output stream, called `stderr`, is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` normally appears on the screen even if the standard output is redirected.

Let us revise `cat` to write its error messages on the standard error.

```
#include <stdio.h>

/* cat: concatenate files, version 2 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0]; /* program name for errors */

    if (argc == 1) /* no args; copy standard input */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(++argv, "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n",
                        prog, *argv);
                exit(1);
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: error writing stdout\n", prog);
        exit(2);
    }
    exit(0);
}
```

The program signals errors two ways. First, the diagnostic output produced by `fprintf` goes onto `stderr`, so it finds its way to the screen instead of disappearing down a pipeline or into an output file. We included the program name, from `argv[0]`, in the message, so if this program is used with others, the source of an error is identified.

Second, the program uses the standard library function `exit`, which terminates program execution when it is called. The argument of `exit` is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process.

Conventionally, a return value of 0 signals that all is well; non-zero values usually signal abnormal situations. `exit` calls `fclose` for each open output file, to flush out any buffered output.

Within `main`, `return expr` is equivalent to `exit(expr)`. `exit` has the advantage that it can be called from other functions, and that calls to it can be found with a pattern-searching program like those in Chapter 5.

The function `ferror` returns non-zero if an error occurred on the stream `fp`.

```
int ferror(FILE *fp)
```

Although output errors are rare, they do occur (for example, if a disk fills up), so a production program should check this as well.

The function `feof(FILE *)` is analogous to `ferror`; it returns non-zero if end of file has occurred on the specified file.

```
int feof(FILE *fp)
```

We have generally not worried about exit status in our small illustrative programs, but any serious program should take care to return sensible, useful status values.

7.7 Line Input and Output

The standard library provides an input routine `fgets` that is similar to the `getline` function that we have used in earlier chapters:

```
char *fgets(char *line, int maxline, FILE *fp)
```

`fgets` reads the next input line (including the newline) from file `fp` into the character array `line`; at most `maxline-1` characters will be read. The resulting line is terminated with `'\0'`. Normally `fgets` returns `line`; on end of file or error it returns `NULL`. (Our `getline` returns the line length, which is a more useful value; zero means end of file.)

For output, the function `fputs` writes a string (which need not contain a newline) to a file:

```
int fputs(char *line, FILE *fp)
```

It returns `EOF` if an error occurs, and zero otherwise.

The library functions `gets` and `puts` are similar to `fgets` and `fputs`, but operate on `stdin` and `stdout`. Confusingly, `gets` deletes the terminal `'\n'`, and `puts` adds it.

To show that there is nothing special about functions like `fgets` and `fputs`, here they are, copied from the standard library on our system:

```

/* fgets:  get at most n chars from iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs:  put string s on file iop */
int fputs(char *s, FILE *iop)
{
    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}

```

The standard specifies that `ferror` returns non-zero for error; `fputs` returns EOF for error and a non-negative value otherwise.

It is easy to implement our `getline` from `fgets`:

```

/* getline:  read a line, return length */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
        return strlen(line);
}

```

Exercise 7-6. Write a program to compare two files, printing the first line where they differ. □

Exercise 7-7. Modify the pattern finding program of Chapter 5 to take its input from a set of named files or, if no files are named as arguments, from the standard input. Should the file name be printed when a matching line is found? □

Exercise 7-8. Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file. □

7.8 Miscellaneous Functions

The standard library provides a wide variety of functions. This section is a brief synopsis of the most useful. More details and many other functions can be found in Appendix B.

7.8.1 String Operations

We have already mentioned the string functions `strlen`, `strcpy`, `strcat`, and `strcmp`, found in `<string.h>`. In the following, `s` and `t` are `char *`'s, and `c` and `n` are `ints`.

<code>strcat(s,t)</code>	concatenate <code>t</code> to end of <code>s</code>
<code>strncat(s,t,n)</code>	concatenate <code>n</code> characters of <code>t</code> to end of <code>s</code>
<code>strcmp(s,t)</code>	return negative, zero, or positive for <code>s < t</code> , <code>s == t</code> , or <code>s > t</code>
<code>strncmp(s,t,n)</code>	same as <code>strcmp</code> but only in first <code>n</code> characters
<code>strcpy(s,t)</code>	copy <code>t</code> to <code>s</code>
<code>strncpy(s,t,n)</code>	copy at most <code>n</code> characters of <code>t</code> to <code>s</code>
<code>strlen(s)</code>	return length of <code>s</code>
<code>strchr(s,c)</code>	return pointer to first <code>c</code> in <code>s</code> , or <code>NULL</code> if not present
<code>strrchr(s,c)</code>	return pointer to last <code>c</code> in <code>s</code> , or <code>NULL</code> if not present

7.8.2 Character Class Testing and Conversion

Several functions from `<ctype.h>` perform character tests and conversions. In the following, `c` is an `int` that can be represented as an unsigned `char`, or `EOF`. The functions return `int`.

<code>isalpha(c)</code>	non-zero if <code>c</code> is alphabetic, 0 if not
<code>isupper(c)</code>	non-zero if <code>c</code> is upper case, 0 if not
<code>islower(c)</code>	non-zero if <code>c</code> is lower case, 0 if not
<code>isdigit(c)</code>	non-zero if <code>c</code> is digit, 0 if not
<code>isalnum(c)</code>	non-zero if <code>isalpha(c)</code> or <code>isdigit(c)</code> , 0 if not
<code>isspace(c)</code>	non-zero if <code>c</code> is blank, tab, newline, return, formfeed, vertical tab
<code>toupper(c)</code>	return <code>c</code> converted to upper case
<code>tolower(c)</code>	return <code>c</code> converted to lower case

7.8.3 Ungetc

The standard library provides a rather restricted version of the function `ungetch` that we wrote in Chapter 4; it is called `ungetc`.

```
int ungetc(int c, FILE *fp)
```

pushes the character `c` back onto file `fp`, and returns either `c`, or `EOF` for an error. Only one character of pushback is guaranteed per file. `ungetc` may be used with any of the input functions like `scanf`, `getc`, or `getchar`.

7.8.4 Command Execution

The function `system(char *s)` executes the command contained in the character string `s`, then resumes execution of the current program. The contents of `s` depend strongly on the local operating system. As a trivial example, on UNIX systems, the statement

```
system("date");
```

causes the program `date` to be run; it prints the date and time of day on the standard output. `system` returns a system-dependent integer status from the command executed. In the UNIX system, the status return is the value returned by `exit`.

7.8.5 Storage Management

The functions `malloc` and `calloc` obtain blocks of memory dynamically.

```
void *malloc(size_t n)
```

returns a pointer to `n` bytes of uninitialized storage, or `NULL` if the request cannot be satisfied.

```
void *calloc(size_t n, size_t size)
```

returns a pointer to enough space for an array of `n` objects of the specified size, or `NULL` if the request cannot be satisfied. The storage is initialized to zero.

The pointer returned by `malloc` or `calloc` has the proper alignment for the object in question, but it must be cast into the appropriate type, as in

```
int *ip;
```

```
ip = (int *) calloc(n, sizeof(int));
```

`free(p)` frees the space pointed to by `p`, where `p` was originally obtained by a call to `malloc` or `calloc`. There are no restrictions on the order in which space is freed, but it is a ghastly error to free something not obtained by calling `calloc` or `malloc`.

It is also an error to use something after it has been freed. A typical but incorrect piece of code is this loop that frees items from a list:

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

The right way is to save whatever is needed before freeing:

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

Section 8.7 shows the implementation of a storage allocator like `malloc`, in

which allocated blocks may be freed in any order.

7.8.6 Mathematical Functions

There are more than twenty mathematical functions declared in `<math.h>`; here are some of the more frequently used. Each takes one or two double arguments and returns a double.

<code>sin(x)</code>	sine of x , x in radians
<code>cos(x)</code>	cosine of x , x in radians
<code>atan2(y,x)</code>	arctangent of y/x , in radians
<code>exp(x)</code>	exponential function e^x
<code>log(x)</code>	natural (base e) logarithm of x ($x > 0$)
<code>log10(x)</code>	common (base 10) logarithm of x ($x > 0$)
<code>pow(x,y)</code>	x^y
<code>sqrt(x)</code>	square root of x ($x \geq 0$)
<code>fabs(x)</code>	absolute value of x

7.8.7 Random Number Generation

The function `rand()` computes a sequence of pseudo-random integers in the range zero to `RAND_MAX`, which is defined in `<stdlib.h>`. One way to produce random floating-point numbers greater than or equal to zero but less than one is

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

(If your library already provides a function for floating-point random numbers, it is likely to have better statistical properties than this one.)

The function `srand(unsigned)` sets the seed for `rand`. The portable implementation of `rand` and `srand` suggested by the standard appears in Section 2.7.

Exercise 7-9. Functions like `isupper` can be implemented to save space or to save time. Explore both possibilities. □