

SECOND EDITION

THE

C



PROGRAMMING
LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Preface to the Digital Edition

The second edition of *The C Programming Language* was published early in 1988. At that time, the first C standard was almost complete, formalizing and codifying the precise definition of the language. There have been two revisions to the standard since then, in 1999 and 2011, that added a number of language features and cleared up a few minor issues. But for many programmers, the 1988 definition of C covers the parts of the language that they use, so it has never seemed necessary to update the book itself to track the newer standards. Thus, the digital version is intentionally identical to the print edition.*

On the other hand, the computing world is very different from what it was in 1988. The Internet has gone from a network primarily for researchers at universities to a universal network linking everyone on the planet. Computers have continued to get smaller, cheaper, and faster; a typical laptop or cell phone today has more computing power than a supercomputer of 1988, yet costs so little that probably half the people in the world have one. Languages such as C++, Objective-C, Java, and JavaScript make it easier to program these systems as well; all of them borrow heavily from C.

Remarkably, in spite of all of this change, C retains a central position. It is still the core language for operating system implementation and tool building. It remains unequalled for portability, efficiency, and ability to get close to the hardware when necessary. C has sometimes been called a high-level assembler, and this is not a bad characterization of how well it spans the range from intricate data structure and control flow to the lowest level of external devices.

Sadly, Dennis Ritchie, the creator of C and the coauthor of this book, died in October 2011 at the age of 70 and never saw this digital edition. Dennis was a great language designer and programmer, and a superb writer, but he was also funny, warm, and exceptionally kind. We are all in his debt. He will be greatly missed.

Brian Kernighan
Princeton, New Jersey
November 2012

* Note: Example code can now be downloaded by visiting www.informit.com/store/c-programming-language-9780131103627.

This page intentionally left blank

THE
C
PROGRAMMING
LANGUAGE

Second Edition

Brian W. Kernighan • Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey



Prentice Hall PTR, Upper Saddle River, New Jersey 07458

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 (pic|tbl|eqn|troff -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.

Fourty-ninth printing, January 2012

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

This page intentionally left blank

Preface

The computing world has undergone a revolution since the publication of *The C Programming Language* in 1978. Big computers are much bigger, and personal computers have capabilities that rival the mainframes of a decade ago. During this time, C has changed too, although only modestly, and it has spread far beyond its origins as the language of the UNIX operating system.

The growing popularity of C, the changes in the language over the years, and the creation of compilers by groups not involved in its design, combined to demonstrate a need for a more precise and more contemporary definition of the language than the first edition of this book provided. In 1983, the American National Standards Institute (ANSI) established a committee whose goal was to produce “an unambiguous and machine-independent definition of the language C,” while still retaining its spirit. The result is the ANSI standard for C.

The standard formalizes constructions that were hinted at but not described in the first edition, particularly structure assignment and enumerations. It provides a new form of function declaration that permits cross-checking of definition with use. It specifies a standard library, with an extensive set of functions for performing input and output, memory management, string manipulation, and similar tasks. It makes precise the behavior of features that were not spelled out in the original definition, and at the same time states explicitly which aspects of the language remain machine-dependent.

This second edition of *The C Programming Language* describes C as defined by the ANSI standard. Although we have noted the places where the language has evolved, we have chosen to write exclusively in the new form. For the most part, this makes no significant difference; the most visible change is the new form of function declaration and definition. Modern compilers already support most features of the standard.

We have tried to retain the brevity of the first edition. C is not a big language, and it is not well served by a big book. We have improved the exposition of critical features, such as pointers, that are central to C programming. We have refined the original examples, and have added new examples in several chapters. For instance, the treatment of complicated declarations is augmented by programs that convert declarations into words and vice versa. As before, all

examples have been tested directly from the text, which is in machine-readable form.

Appendix A, the reference manual, is not the standard, but our attempt to convey the essentials of the standard in a smaller space. It is meant for easy comprehension by programmers, but not as a definition for compiler writers—that role properly belongs to the standard itself. Appendix B is a summary of the facilities of the standard library. It too is meant for reference by programmers, not implementers. Appendix C is a concise summary of the changes from the original version.

As we said in the preface to the first edition, C “wears well as one’s experience with it grows.” With a decade more experience, we still feel that way. We hope that this book will help you to learn C and to use it well.

We are deeply indebted to friends who helped us to produce this second edition. Jon Bentley, Doug Gwyn, Doug McIlroy, Peter Nelson, and Rob Pike gave us perceptive comments on almost every page of draft manuscripts. We are grateful for careful reading by Al Aho, Dennis Allison, Joe Campbell, G. R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford, and Chris Van Wyk. We also received helpful suggestions from Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo, and Peter Weinberger. Dave Prosser answered many detailed questions about the ANSI standard. We used Bjarne Stroustrup’s C++ translator extensively for local testing of our programs, and Dave Kristol provided us with an ANSI C compiler for final testing. Rich Drechsler helped greatly with typesetting.

Our sincere thanks to all.

Brian W. Kernighan
Dennis M. Ritchie

Preface to the First Edition

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C is not a “very high level” language, nor a “big” one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs (including all of the software used to prepare this book) are written in C. Production compilers also exist for several other machines, including the IBM System/370, the Honeywell 6000, and the Interdata 8/32. C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C.

This book is meant to help the reader learn how to program in C. It contains a tutorial introduction to get new users started as soon as possible, separate chapters on each major feature, and a reference manual. Most of the treatment is based on reading, writing and revising examples, rather than on mere statements of rules. For the most part, the examples are complete, real programs, rather than isolated fragments. All examples have been tested directly from the text, which is in machine-readable form. Besides showing how to make effective use of the language, we have also tried where possible to illustrate useful algorithms and principles of good style and sound design.

The book is not an introductory programming manual; it assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and functions. Nonetheless, a novice programmer should be able to read along and pick up the language, although access to a more knowledgeable colleague will help.

In our experience, C has proven to be a pleasant, expressive, and versatile language for a wide variety of programs. It is easy to learn, and it wears well as one’s experience with it grows. We hope that this book will help you to use it well.

The thoughtful criticisms and suggestions of many friends and colleagues have added greatly to this book and to our pleasure in writing it. In particular, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin, and Larry Rosler all read multiple versions with care. We are also indebted to Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson, and Peter Weinberger for helpful comments at various stages, and to Mike Lesk and Joe Ossanna for invaluable assistance with typesetting.

Brian W. Kernighan
Dennis M. Ritchie

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 3: Control Flow

The control-flow statements of a language specify the order in which computations are performed. We have already met the most common control-flow constructions in earlier examples; here we will complete the set, and be more precise about the ones discussed before.

3.1 Statements and Blocks

An expression such as `x = 0` or `i++` or `printf(...)` becomes a *statement* when it is followed by a semicolon, as in

```
x = 0;  
i++;  
printf(...);
```

In C, the semicolon is a statement terminator, rather than a separator as it is in languages like Pascal.

Braces `{` and `}` are used to group declarations and statements together into a *compound statement*, or *block*, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an `if`, `else`, `while`, or `for` are another. (Variables can be declared inside *any* block; we will talk about this in Chapter 4.) There is no semicolon after the right brace that ends a block.

3.2 If-Else

The `if-else` statement is used to express decisions. Formally, the syntax is

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

where the `else` part is optional. The *expression* is evaluated; if it is true (that is, if *expression* has a non-zero value), *statement*₁ is executed. If it is false (*expression* is zero) and if there is an `else` part, *statement*₂ is executed instead.

Since an `if` simply tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

```
if (expression)
```

instead of

```
if (expression != 0)
```

Sometimes this is natural and clear; at other times it can be cryptic.

Because the `else` part of an `if-else` is optional, there is an ambiguity when an `else` is omitted from a nested `if` sequence. This is resolved by associating the `else` with the closest previous `else-less if`. For example, in

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

the `else` goes with the inner `if`, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

The ambiguity is especially pernicious in situations like this:

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else /* WRONG */
    printf("error -- n is negative\n");
```

The indentation shows unequivocally what you want, but the compiler doesn't get the message, and associates the `else` with the inner `if`. This kind of bug can be hard to find; it's a good idea to use braces when there are nested `ifs`.

By the way, notice that there is a semicolon after `z = a` in

```

if (a > b)
    z = a;
else
    z = b;

```

This is because grammatically, a *statement* follows the *if*, and an expression statement like “*z = a;*” is always terminated by a semicolon.

3.3 Else-If

The construction

```

if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else
    statement

```

occurs so often that it is worth a brief separate discussion. This sequence of *if* statements is the most general way of writing a multi-way decision. The *expressions* are evaluated in order; if any *expression* is true, the *statement* associated with it is executed, and this terminates the whole chain. As always, the code for each *statement* is either a single statement, or a group in braces.

The last *else* part handles the “none of the above” or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

```

else
    statement

```

can be omitted, or it may be used for error checking to catch an “impossible” condition.

To illustrate a three-way decision, here is a binary search function that decides if a particular value *x* occurs in the sorted array *v*. The elements of *v* must be in increasing order. The function returns the position (a number between 0 and *n*-1) if *x* occurs in *v*, and -1 if not.

Binary search first compares the input value *x* to the middle element of the array *v*. If *x* is less than the middle value, searching focuses on the lower half of the table, otherwise on the upper half. In either case, the next step is to compare *x* to the middle element of the selected half. This process of dividing the range in two continues until the value is found or the range is empty.

```

/* binsearch: find x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else /* found match */
            return mid;
    }
    return -1; /* no match */
}

```

The fundamental decision is whether x is less than, greater than, or equal to the middle element $v[\text{mid}]$ at each step; this is a natural for `else-if`.

Exercise 3-1. Our binary search makes two tests inside the loop, when one would suffice (at the price of more tests outside). Write a version with only one test inside the loop and measure the difference in run-time. □

3.4 Switch

The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of *constant* integer values, and branches accordingly.

```

switch (expression) {
    case const-expr: statements
    case const-expr: statements
    default: statements
}

```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled `default` is executed if none of the other cases are satisfied. A `default` is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

In Chapter 1 we wrote a program to count the occurrences of each digit, white space, and all other characters, using a sequence of `if ... else if ... else`. Here is the same program with a `switch`:

```

#include <stdio.h>

main() /* count digits, white space, others */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF) {
        switch (c) {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':
                ndigit[c-'0']++;
                break;
            case ' ':
            case '\n':
            case '\t':
                nwhite++;
                break;
            default:
                nother++;
                break;
        }
    }
    printf("digits =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
        nwhite, nother);
    return 0;
}

```

The `break` statement causes an immediate exit from the `switch`. Because cases serve just as labels, after the code for one case is done, execution *falls through* to the next unless you take explicit action to escape. `break` and `return` are the most common ways to leave a `switch`. A `break` statement can also be used to force an immediate exit from `while`, `for`, and `do` loops, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits in this example. But it also implies that normally each case must end with a `break` to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly, and commented.

As a matter of good form, put a `break` after the last case (the `default` here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

Exercise 3-2. Write a function `escape(s,t)` that converts characters like newline and tab into visible escape sequences like `\n` and `\t` as it copies the string `t` to `s`. Use a `switch`. Write a function for the other direction as well, converting escape sequences into the real characters. □

3.5 Loops—While and For

We have already encountered the `while` and `for` loops. In

```
while (expression)
    statement
```

the *expression* is evaluated. If it is non-zero, *statement* is executed and *expression* is re-evaluated. This cycle continues until *expression* becomes zero, at which point execution resumes after *statement*.

The `for` statement

```
for (expr1; expr2; expr3)
    statement
```

is equivalent to

```
expr1;
while (expr2) {
    statement
    expr3;
}
```

except for the behavior of `continue`, which is described in Section 3.7.

Grammatically, the three components of a `for` loop are expressions. Most commonly, `expr1` and `expr3` are assignments or function calls and `expr2` is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If `expr1` or `expr3` is omitted, it is simply dropped from the expansion. If the test, `expr2`, is not present, it is taken as permanently true, so

```
for (;;) {
    ...
}
```

is an “infinite” loop, presumably to be broken by other means, such as a `break` or `return`.

Whether to use `while` or `for` is largely a matter of personal preference. For example, in

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ; /* skip white space characters */
```

there is no initialization or re-initialization, so the `while` is most natural.

The `for` is preferable when there is a simple initialization and increment, since it keeps the loop control statements close together and visible at the top of

the loop. This is most obvious in

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

which is the C idiom for processing the first n elements of an array, the analog of the Fortran DO loop or the Pascal `for`. The analogy is not perfect, however, since the index and limit of a C `for` loop can be altered from within the loop, and the index variable `i` retains its value when the loop terminates for any reason. Because the components of the `for` are arbitrary expressions, `for` loops are not restricted to arithmetic progressions. Nonetheless, it is bad style to force unrelated computations into the initialization and increment of a `for`, which are better reserved for loop control operations.

As a larger example, here is another version of `atoi` for converting a string to its numeric equivalent. This one is slightly more general than the one in Chapter 2; it copes with optional leading white space and an optional `+` or `-` sign. (Chapter 4 shows `atof`, which does the same conversion for floating-point numbers.)

The structure of the program reflects the form of the input:

```
skip white space, if any
get sign, if any
get integer part and convert it
```

Each step does its part, and leaves things in a clean state for the next. The whole process terminates on the first character that could not be part of a number.

```
#include <ctype.h>

/* atoi: convert s to integer; version 2 */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++) /* skip white space */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-') /* skip sign */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

The standard library provides a more elaborate function `strtol` for conversion of strings to long integers; see Section 5 of Appendix B.

The advantages of keeping loop control centralized are even more obvious when there are several nested loops. The following function is a Shell sort for sorting an array of integers. The basic idea of this sorting algorithm, which was

invented in 1959 by D. L. Shell, is that in early stages, far-apart elements are compared, rather than adjacent ones as in simpler interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```

/* shellsort: sort v[0]...v[n-1] into increasing order */
void shellsort(int v[], int n)
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}

```

There are three nested loops. The outermost controls the gap between compared elements, shrinking it from $n/2$ by a factor of two each pass until it becomes zero. The middle loop steps along the elements. The innermost loop compares each pair of elements that is separated by `gap` and reverses any that are out of order. Since `gap` is eventually reduced to one, all elements are eventually ordered correctly. Notice how the generality of the `for` makes the outer loop fit the same form as the others, even though it is not an arithmetic progression.

One final C operator is the comma “,”, which most often finds use in the `for` statement. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand. Thus in a `for` statement, it is possible to place multiple expressions in the various parts, for example to process two indices in parallel. This is illustrated in the function `reverse(s)`, which reverses the string `s` in place.

```

#include <string.h>

/* reverse: reverse string s in place */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

```

The commas that separate function arguments, variables in declarations, etc., are *not* comma operators, and do not guarantee left to right evaluation.

Comma operators should be used sparingly. The most suitable uses are for constructs strongly related to each other, as in the `for` loop in `reverse`, and in macros where a multistep computation has to be a single expression. A comma expression might also be appropriate for the exchange of elements in `reverse`, where the exchange can be thought of as a single operation:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

Exercise 3-3. Write a function `expand(s1,s2)` that expands shorthand notations like `a-z` in the string `s1` into the equivalent complete list `abc...xyz` in `s2`. Allow for letters of either case and digits, and be prepared to handle cases like `a-b-c` and `a-z0-9` and `-a-z`. Arrange that a leading or trailing `-` is taken literally. □

3.6 Loops—Do-while

As we discussed in Chapter 1, the `while` and `for` loops test the termination condition at the top. By contrast, the third loop in C, the `do-while`, tests at the bottom *after* making each pass through the loop body; the body is always executed at least once.

The syntax of the `do` is

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

The *statement* is executed, then *expression* is evaluated. If it is true, *statement* is evaluated again, and so on. When the expression becomes false, the loop terminates. Except for the sense of the test, `do-while` is equivalent to the Pascal `repeat-until` statement.

Experience shows that `do-while` is much less used than `while` and `for`. Nonetheless, from time to time it is valuable, as in the following function `itoa`, which converts a number to a character string (the inverse of `atoi`). The job is slightly more complicated than might be thought at first, because the easy methods of generating the digits generate them in the wrong order. We have chosen to generate the string backwards, then reverse it.

```

/* itoa: convert n to characters in s */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0) /* record sign */
        n = -n;      /* make n positive */
    i = 0;
    do { /* generate digits in reverse order */
        s[i++] = n % 10 + '0'; /* get next digit */
    } while ((n /= 10) > 0); /* delete it */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}

```

The `do-while` is necessary, or at least convenient, since at least one character must be installed in the array `s`, even if `n` is zero. We also used braces around the single statement that makes up the body of the `do-while`, even though they are unnecessary, so the hasty reader will not mistake the `while` part for the *beginning* of a `while` loop.

Exercise 3-4. In a two's complement number representation, our version of `itoa` does not handle the largest negative number, that is, the value of `n` equal to $-(2^{\text{wordsize}-1})$. Explain why not. Modify it to print that value correctly, regardless of the machine on which it runs. □

Exercise 3-5. Write the function `itob(n,s,b)` that converts the integer `n` into a base `b` character representation in the string `s`. In particular, `itob(n,s,16)` formats `n` as a hexadecimal integer in `s`. □

Exercise 3-6. Write a version of `itoa` that accepts three arguments instead of two. The third argument is a minimum field width; the converted number must be padded with blanks on the left if necessary to make it wide enough. □

3.7 Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The `break` statement provides an early exit from `for`, `while`, and `do`, just as from `switch`. A `break` causes the innermost enclosing loop or `switch` to be exited immediately.

The following function, `trim`, removes trailing blanks, tabs, and newlines from the end of a string, using a `break` to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.

```

/* trim: remove trailing blanks, tabs, newlines */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}

```

`strlen` returns the length of the string. The `for` loop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found, or when `n` becomes negative (that is, when the entire string has been scanned). You should verify that this is correct behavior even when the string is empty or contains only white space characters.

The `continue` statement is related to `break`, but less often used; it causes the next iteration of the enclosing `for`, `while`, or `do` loop to begin. In the `while` and `do`, this means that the test part is executed immediately; in the `for`, control passes to the increment step. The `continue` statement applies only to loops, not to `switch`. A `continue` inside a `switch` inside a loop causes the next loop iteration.

As an example, this fragment processes only the non-negative elements in the array `a`; negative values are skipped.

```

for (i = 0; i < n; i++) {
    if (a[i] < 0) /* skip negative elements */
        continue;
    ... /* do positive elements */
}

```

The `continue` statement is often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

3.8 Goto and Labels

`C` provides the infinitely-abusable `goto` statement, and labels to branch to. Formally, the `goto` is never necessary, and in practice it is almost always easy to write code without it. We have not used `goto` in this book.

Nevertheless, there are a few situations where `gotos` may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The `break` statement cannot be used directly since it only exits from the innermost loop. Thus:

```

    for ( ... )
        for ( ... ) {
            ...
            if (disaster)
                goto error;
        }
    ...

error:
    clean up the mess

```

This organization is handy if the error-handling code is non-trivial, and if errors can occur in several places.

A label has the same form as a variable name, and is followed by a colon. It can be attached to any statement in the same function as the `goto`. The scope of a label is the entire function.

As another example, consider the problem of determining whether two arrays `a` and `b` have an element in common. One possibility is

```

    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if (a[i] == b[j])
                goto found;
    /* didn't find any common element */
    ...
found:
    /* got one: a[i] == b[j] */
    ...

```

Code involving a `goto` can always be written without one, though perhaps at the price of some repeated tests or an extra variable. For example, the array search becomes

```

found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;
if (found)
    /* got one: a[i-1] == b[j-1] */
    ...
else
    /* didn't find any common element */
    ...

```

With a few exceptions like those cited here, code that relies on `goto` statements is generally harder to understand and to maintain than code without `gotos`. Although we are not dogmatic about the matter, it does seem that `goto` statements should be used rarely, if at all.

Index

- 0... octal constant 37, 193
- 0x... hexadecimal constant 37, 193
- + addition operator 41, 205
- & address operator 93, 203
- = assignment operator 17, 42, 208
- += assignment operator 50
- \ backslash character 8, 38
- & bitwise AND operator 48, 207
- ^ bitwise exclusive OR operator 48, 207
- ! bitwise inclusive OR operator 48, 207
- , comma operator 62, 209
- ? : conditional expression 51, 208
- ... declaration 155, 202
- decrement operator 18, 46, 106, 203
- / division operator 10, 41, 205
- == equality operator 19, 41, 207
- >= greater or equal operator 41, 206
- > greater than operator 41, 206
- ++ increment operator 18, 46, 106, 203
- * indirection operator 94, 203
- != inequality operator 16, 41, 207
- << left shift operator 49, 206
- <= less or equal operator 41, 206
- < less than operator 41, 206
- && logical AND operator 21, 41, 49, 207
- ! logical negation operator 42, 203–204
- || logical OR operator 21, 41, 49, 208
- % modulus operator 41, 205
- * multiplication operator 41, 205
- one's complement operator 49, 203–204
- # preprocessor operator 90, 230
- ## preprocessor operator 90, 230
- ' quote character 19, 37–38, 193
- " quote character 8, 20, 38, 194
- >> right shift operator 49, 206
- . structure member operator 128, 201
- > structure pointer operator 131, 201
- subtraction operator 41, 205
- unary minus operator 203–204
- + unary plus operator 203–204
- _ underscore character 35, 192, 241
- \0 null character 30, 38, 193

- \a alert character 38, 193
- abort library function 252

- abs library function 253
- abstract declarator 220
- access mode, file 160, 178, 242
- acos library function 251
- actual argument *see* argument
- addition operator, + 41, 205
- additive operators 205
- addpoint function 130
- address arithmetic *see* pointer arithmetic
- address of register 210
- address of variable 28, 94, 203
- address operator, & 93, 203
- addtree function 141
- afree function 102
- alert character, \a 38, 193
- alignment, bit-field 150, 213
- alignment by union 186
- alignment restriction 138, 142, 148, 167, 185, 199
- alloc function 101
- allocator, storage 142, 185–189
- ambiguity, if-else 56, 223, 234
- American National Standards Institute (ANSI) ix, 2, 191
- a.out 6, 70
- argc argument count 114
- argument, definition of 25, 201
- argument, function 25, 202
- argument list, variable length 155, 174, 202, 218, 225, 254
- argument list, void 33, 73, 218, 225
- argument, pointer 100
- argument promotion 45, 202
- argument, subarray 100
- arguments, command-line 114–118
- argv argument vector 114, 163
- arithmetic conversions, usual 42, 198
- arithmetic operators 41
- arithmetic, pointer 94, 98, 100–103, 117, 138, 205
- arithmetic types 196
- array, character 20, 28, 104
- array declaration 22, 111, 216
- array declarator 216
- array initialization 86, 113, 219
- array, initialization of two-dimensional 112, 220

- array, multi-dimensional 110, 217
- array name argument 28, 100, 112
- array name, conversion of 99, 200
- array of pointers 107
- array reference 201
- array size, default 86, 113, 133
- array, storage order of 112, 217
- array subscripting 22, 97, 201, 217
- array, two-dimensional 110, 112, 220
- array vs. pointer 97, 99–100, 104, 113
- arrays of structures 132
- ASCII character set 19, 37, 43, 229, 249
- asctime library function 256
- asin library function 251
- asm keyword 192
- <assert.h> header 253
- assignment, conversion by 44, 208
- assignment expression 17, 21, 51, 208
- assignment, multiple 21
- assignment operator, = 17, 42, 208
- assignment operator, += 50
- assignment operators 42, 50, 208
- assignment statement, nested 17, 21, 51
- assignment suppression, scanf 157, 245
- associativity of operators 52, 200
- atan, atan2 library functions 251
- atexit library function 253
- atof function 71
- atof library function 251
- atoi function 43, 61, 73
- atoi library function 251
- atol library function 251
- auto storage class specifier 210
- automatic storage class 31, 195
- automatic variable 31, 74, 195
- automatics, initialization of 31, 40, 85, 219
- automatics, scope of 80, 228
- avoiding goto 66

- \b backspace character 8, 38, 193
- backslash character, \\ 8, 38
- bell character *see* alert character
- binary stream 160, 241–242
- binary tree 139
- binsearch function 58, 134, 137
- bit manipulation idioms 49, 149
- bitcount function 50
- bit-field alignment 150, 213
- bit-field declaration 150, 212
- bitwise AND operator, & 48, 207
- bitwise exclusive OR operator, ^ 48, 207
- bitwise inclusive OR operator, | 48, 207
- bitwise operators 48, 207
- block *see* compound statement
- block, initialization in 84, 223
- block structure 55, 84, 223
- boundary condition 19, 65
- braces 7, 10, 55, 84
- braces, position of 10
- break statement 59, 64, 224
- bsearch library function 253
- buffered getchar 172
- buffered input 170

- buffering *see* setbuf, setvbuf
- BUFSIZ 243

- calculator program 72, 74, 76, 158
- call by reference 27
- call by value 27, 95, 202
- calloc library function 167, 252
- canonrect function 131
- carriage return character, \r 38, 193
- case label 58, 222
- cast, conversion by 45, 198–199, 205
- cast operator 45, 142, 167, 198, 205, 220
- cat program 160, 162–163
- cc command 6, 70
- ceil library function 251
- char type 10, 36, 195, 211
- character array 20, 28, 104
- character constant 19, 37, 193
- character constant, octal 37
- character constant, wide 193
- character count program 18
- character input/output 15, 151
- character set 229
- character set, ASCII 19, 37, 43, 229, 249
- character set, EBCDIC 43
- character set, ISO 229
- character, signed 44, 195
- character string *see* string constant
- character testing functions 166, 248
- character, unsigned 44, 195
- character-integer conversion 23, 42, 197
- characters, white space 157, 166, 245, 249
- clearerr library function 248
- CLOCKS_PER_SEC 255
- clock library function 255
- clock_t type name 255
- close system call 174
- closedir function 184
- coercion *see* cast
- comma operator, , 62, 209
- command-line arguments 114–118
- comment 9, 191–192, 229
- comparison, pointer 102, 138, 187, 207
- compilation, separate 67, 80, 227
- compiling a C program 6, 25
- compiling multiple files 70
- compound statement 55, 84, 222, 225–226
- concatenation, string 38, 90, 194
- concatenation, token 90, 230
- conditional compilation 91, 231
- conditional expression, ?: 51, 208
- const qualifier 40, 196, 211
- constant expression 38, 58, 91, 209
- constant, manifest 230
- constant suffix 37, 193
- constant, type of 37, 193
- constants 37, 192
- continue statement 65, 224
- control character 249
- control line 88, 229–233
- conversion 197–199
- conversion by assignment 44, 208
- conversion by cast 45, 198–199, 205

- conversion by `return` 73, 225
- conversion, character-integer 23, 42, 197
- conversion, `double-float` 45, 198
- conversion, `float-double` 44, 198
- conversion, floating-integer 45, 197
- conversion, integer-character 45
- conversion, integer-floating 12, 197
- conversion, integer-pointer 199, 205
- conversion of array name 99, 200
- conversion of function 200
- conversion operator, explicit *see* `cast`
- conversion, pointer 142, 198, 205
- conversion, pointer-integer 198–199, 205
- conversions, usual arithmetic 42, 198
- `copy` function 29, 33
- `cos` library function 251
- `cosh` library function 251
- `creat` system call 172
- `CRLF` 151, 241
- `ctime` library function 256
- `<ctype.h>` header 43, 248

- date conversion 111
- `day_of_year` function 111
- `dcl` function 123
- `dcl` program 125
- declaration 9, 40, 210–218
- declaration, array 22, 111, 216
- declaration, bit-field 150, 212
- declaration, external 225–226
- declaration of external variable 31, 225
- declaration of function 217–218
- declaration of function, implicit 27, 72, 201
- declaration of pointer 94, 100, 216
- declaration, storage class 210
- declaration, structure 128, 212
- declaration, type 216
- declaration, `typedef` 146, 210, 221
- declaration, union 147, 212
- declaration vs. definition 33, 80, 210
- declarator 215–218
- declarator, abstract 220
- declarator, array 216
- declarator, function 217
- decrement operator, `--` 18, 46, 106, 203
- default array size 86, 113, 133
- default function type 30, 201
- default initialization 86, 219
- default label 58, 222
- defensive programming 57, 59
- `#define` 14, 89, 229
- `#define`, multi-line 89
- `#define` vs. `enum` 39, 149
- `#define` with arguments 89
- `defined` preprocessor operator 91, 232
- definition, function 25, 69, 225
- definition, macro 229
- definition of argument 25, 201
- definition of external variable 33, 227
- definition of parameter 25, 201
- definition of storage 210
- definition, removal of *see* `#undef`
- definition, tentative 227

- dereference *see* indirection
- derived types 1, 10, 196
- descriptor, file 170
- designator, function 201
- `difftime` library function 256
- `DIR` structure 180
- `dirfd` function 124
- directory list program 179
- `Dirent` structure 180
- `dir.h` include file 183
- `dirwalk` function 182
- `div` library function 253
- division, integer 10, 41
- division operator, `/` 10, 41, 205
- `div_t`, `ldiv_t` type names 253
- `do` statement 63, 224
- `do-nothing` function 70
- double constant 37, 194
- double type 10, 18, 36, 196, 211
- double-float conversion 45, 198

- E notation 37, 194
- EBCDIC character set 43
- `echo` program 115–116
- EDOM 250
- efficiency 51, 83, 88, 142, 187
- `else` *see* `if-else` statement
- `#else`, `#elif` 91, 232
- `else-if` 23, 57
- empty function 70
- empty statement *see* `null` statement
- empty string 38
- end of file *see* `EOF`
- `#endif` 91
- `enum` specifier 39, 215
- `enum` vs. `#define` 39, 149
- enumeration constant 39, 91, 193–194, 215
- enumeration tag 215
- enumeration type 196
- enumerator 194, 215
- `EOF` 16, 151, 242
- equality operator, `==` 19, 41, 207
- equality operators 41, 207
- equivalence, type 221
- `ERANGE` 250
- `errno` 248, 250
- `<errno.h>` header 248
- `#error` 233
- `error` function 174
- errors, input/output 164, 248
- escape sequence 8, 19, 37–38, 193, 229
- escape sequence, `\x` hexadecimal 37, 193
- escape sequences, table of 38, 193
- evaluation, order of 21, 49, 53, 63, 77, 90, 95, 200
- exceptions 200, 255
- `exit` library function 163, 252
- `EXIT_FAILURE`, `EXIT_SUCCESS` 252
- `exp` library function 251
- expansion, macro 230
- explicit conversion operator *see* `cast`
- exponentiation 24, 251
- expression 200–209

- expression, assignment 17, 21, 51, 208
- expression, constant 38, 58, 91, 209
- expression order of evaluation 52, 200
- expression, parenthesized 201
- expression, primary 200
- expression statement 55, 57, 222
- extern storage class specifier 31, 33, 80, 210
- external declaration 225–226
- external linkage 73, 192, 195, 211, 228
- external names, length of 35, 192
- external static variables 83
- external variable 31, 73, 195
- external variable, declaration of 31, 225
- external variable, definition of 33, 227
- externals, initialization of 40, 81, 85, 219
- externals, scope of 80, 228

- \f formfeed character 38, 193
- fabs library function 251
- fclose library function 162, 242
- fcntl.h include file 172
- feof library function 164, 248
- feof macro 176
- ferror library function 164, 248
- ferror macro 176
- fflush library function 242
- fgetc library function 246
- fgetpos library function 248
- fgets function 165
- fgets library function 164, 247
- field *see* bit-field
- file access 160, 169, 178, 242
- file access mode 160, 178, 242
- file appending 160, 175, 242
- file concatenation program 160
- file copy program 16–17, 171, 173
- file creation 161, 169
- file descriptor 170
- file inclusion 88, 231
- file opening 160, 169, 172
- file permissions 173
- file pointer 160, 175, 242
- __FILE__ preprocessor name 254
- FILE type name 160
- filecopy function 162
- filename suffix, .h 33
- FILENAME_MAX 242
- _fillbuf function 178
- float constant 37, 194
- float type 9, 36, 196, 211
- float-double conversion 44, 198
- <float.h> header 36, 257
- floating constant 12, 37, 194
- floating point, truncation of 45, 197
- floating types 196
- floating-integer conversion 45, 197
- floor library function 251
- fmod library function 251
- fopen function 177
- fopen library function 160, 242
- FOPEN_MAX 242
- for(;;) infinite loop 60, 89
- for statement 13, 18, 60, 224

- for vs while 14, 60
- formal parameter *see* parameter
- formatted input *see* scanf
- formatted output *see* printf
- formfeed character, \f 38, 193
- fortran keyword 192
- fpos_t type name 248
- fprintf library function 161, 243
- fputc library function 247
- fputs function 165
- fputs library function 164, 247
- fread library function 247
- free function 188
- free library function 167, 252
- freopen library function 162, 242
- frexp library function 251
- fscanf library function 161, 245
- fseek library function 248
- fsetpos library function 248
- fsz function 182
- fsz program 181
- fstat system call 183
- ftell library function 248
- function argument 25, 202
- function argument conversion *see* argument promotion
- function call semantics 201
- function call syntax 201
- function, conversion of 200
- function, declaration of 217–218
- function declaration, static 83
- function declarator 217
- function definition 25, 69, 225
- function designator 201
- function, implicit declaration of 27, 72, 201
- function names, length of 35, 192
- function, new-style 202
- function, old-style 26, 33, 72, 202
- function, pointer to 118, 147, 201
- function prototype 26, 30, 45, 72, 120, 202
- function type, default 30, 201
- functions, character testing 166, 248
- fundamental types 9, 36, 195
- fwrite library function 247

- generic pointer *see* void * pointer
- getbits function 49
- getc library function 161, 247
- getc macro 176
- getch function 79
- getchar, buffered 172
- getchar library function 15, 151, 161, 247
- getchar, unbuffered 171
- getenv library function 253
- getint function 97
- getline function 29, 32, 69, 165
- getop function 78
- gets library function 164, 247
- gettoken function 125
- getword function 136
- gmtime library function 256
- goto statement 65, 224
- greater or equal operator, >= 41, 206

- greater than operator, > 41, 206
- .h filename suffix 33
- hash function 144
- hash table 144
- header file 33, 82
- headers, table of standard 241
- hexadecimal constant, 0x... 37, 193
- hexadecimal escape sequence, \x 37, 193
- Hoare, C. A. R. 87
- HUGE_VAL 250
- identifier 192
- #if 91, 135, 231
- #ifdef 91, 232
- if-else ambiguity 56, 223, 234
- if-else statement 19, 21, 55, 223
- #ifndef 91, 232
- illegal pointer arithmetic 102–103, 138, 205
- implicit declaration of function 27, 72, 201
- #include 33, 88, 152, 231
- incomplete type 212
- inconsistent type declaration 72
- increment operator, ++ 18, 46, 106, 203
- indentation 10, 19, 23, 56
- indirection operator, * 94, 203
- inequality operator, != 16, 41, 207
- infinite loop, for(;;) 60, 89
- information hiding 67–68, 75, 77
- initialization 40, 85, 218
- initialization, array 86, 113, 219
- initialization by string constant 86, 219
- initialization, default 86, 219
- initialization in block 84, 223
- initialization of automatics 31, 40, 85, 219
- initialization of externals 40, 81, 85, 219
- initialization of statics 40, 85, 219
- initialization of structure arrays 133
- initialization of two-dimensional array 112, 220
- initialization, pointer 102, 138
- initialization, structure 128, 219
- initialization, union 219
- initializer 227
- initializer, form of 85, 209
- inode 179
- input, buffered 170
- input, formatted *see* scanf
- input, keyboard 15, 151, 170
- input pushback 78
- input, unbuffered 170
- input/output, character 15, 151
- input/output errors 164, 248
- input/output redirection 152, 161, 170
- install function 145
- int type 9, 36, 211
- integer constant 12, 37, 193
- integer-character conversion 45
- integer-floating conversion 12, 197
- integer-pointer conversion 199, 205
- integral promotion 44, 197
- integral types 196
- internal linkage 195, 228
- internal names, length of 35, 192
- internal static variables 83
- _IOFBF, _IOLBF, _IONBF 243
- isalnum library function 136, 249
- isalpha library function 136, 166, 249
- iscntrl library function 249
- isdigit library function 166, 249
- isgraph library function 249
- islower library function 166, 249
- ISO character set 229
- isprint library function 249
- ispunct library function 249
- isspace library function 136, 166, 249
- isupper library function 166, 249
- isxdigit library function 249
- iteration statements 224
- itoa function 64
- jump statements 224
- keyboard input 15, 151, 170
- keyword count program 133
- keywords, list of 192
- label 65, 222
- label, case 58, 222
- label, default 58, 222
- label, scope of 66, 222, 228
- labeled statement 65, 222
- labs library function 253
- %ld conversion 18
- ldexp library function 251
- ldiv library function 253
- leap year computation 41, 111
- left shift operator, << 49, 206
- length of names 35, 192
- length of string 30, 38, 104
- length of variable names 192
- less or equal operator, <= 41, 206
- less than operator, < 41, 206
- lexical conventions 191
- lexical scope 227
- lexicographic sorting 118
- library function 7, 67, 80
- <limits.h> header 36, 257
- #line 233
- line count program 19
- __LINE__ preprocessor name 254
- line splicing 229
- linkage 195, 227–228
- linkage, external 73, 192, 195, 211, 228
- linkage, internal 195, 228
- list directory program 179
- list of keywords 192
- locale issues 241
- <locale.h> header 241
- localtime library function 256
- log, log₁₀ library functions 251
- logical AND operator, && 21, 41, 49, 207
- logical expression, numeric value of 44

- logical negation operator, ! 42, 203–204
- logical OR operator, || 21, 41, 49, 208
- long constant 37, 193
- long double constant 37, 194
- long double type 36, 196
- long type 10, 18, 36, 196, 211
- longest-line program 29, 32
- longjmp library function 254
- LONG_MAX, LONG_MIN 252
- lookup function 145
- loop *see* while, for, do
- lower case conversion program 153
- lower function 43
- ls command 179
- lseek system call 174
- lvalue 197

- macro preprocessor 88, 228–233
- macros with arguments 89
- magic numbers 14
- main function 6
- main, return from 26, 164
- makepoint function 130
- malloc function 187
- malloc library function 143, 167, 252
- manifest constant 230
- <math.h> header 44, 250
- member name, structure 128, 213
- memchr library function 250
- memcmp library function 250
- memcpy library function 250
- memmove library function 250
- memset library function 250
- missing storage class specifier 211
- missing type specifier 211
- mkttime library function 256
- modf library function 251
- modularization 24, 28, 34, 67, 74–75, 108
- modulus operator, % 41, 205
- month_day function 111
- month_name function 113
- morecore function 188
- multi-dimensional array 110, 217
- multiple assignment 21
- multiple files, compiling 70
- multiplication operator, * 41, 205
- multiplicative operators 205
- multi-way decision 23, 57
- mutually recursive structures 140, 213

- \n newline character 7, 15, 20, 37–38, 193, 241
- name 192
- name hiding 84
- name space 227
- names, length of 35, 192
- negative subscripts 100
- nested assignment statement 17, 21, 51
- nested structure 129
- newline 191, 229
- newline character, \n 7, 15, 20, 37–38, 193, 241

- new-style function 202
- NULL 102
- null character, \0 30, 38, 193
- null pointer 102, 198
- null statement 18, 222
- null string 38
- numbers, size of 9, 18, 36, 257
- numcmp function 121
- numeric sorting 118
- numeric value of logical expression 44
- numeric value of relational expression 42, 44

- object 195, 197
- octal character constant 37
- octal constant, 0... 37, 193
- old-style function 26, 33, 72, 202
- one's complement operator, ~ 49, 203–204
- open system call 172
- opendir function 183
- operations on unions 148
- operations permitted on pointers 103
- operators, additive 205
- operators, arithmetic 41
- operators, assignment 42, 50, 208
- operators, associativity of 52, 200
- operators, bitwise 48, 207
- operators, equality 41, 207
- operators, multiplicative 205
- operators, precedence of 17, 52, 95, 131–132, 200
- operators, relational 16, 41, 206
- operators, shift 48, 206
- operators, table of 53
- order of evaluation 21, 49, 53, 63, 77, 90, 95, 200
- order of translation 228
- O_RDONLY, O_RDWR, O_WRONLY 172
- output, formatted *see* printf
- output redirection 152
- output, screen 15, 152, 163, 170
- overflow 41, 200, 250, 255

- parameter 84, 99, 202
- parameter, definition of 25, 201
- parenthesized expression 201
- parse tree 123
- parser, recursive-descent 123
- pattern finding program 67, 69, 116–117
- permissions, file 173
- perror library function 248
- phases, translation 191, 228
- pipe 152, 170
- pointer argument 100
- pointer arithmetic 94, 98, 100–103, 117, 138, 205
- pointer arithmetic, illegal 102–103, 138, 205
- pointer arithmetic, scaling in 103, 198
- pointer comparison 102, 138, 187, 207
- pointer conversion 142, 198, 205
- pointer, declaration of 94, 100, 216
- pointer, file 160, 175, 242
- pointer generation 200

- pointer initialization 102, 138
- pointer, null 102, 198
- pointer subtraction 103, 138, 198
- pointer to function 118, 147, 201
- pointer to structure 136
- pointer, void * 93, 103, 120, 199
- pointer vs. array 97, 99–100, 104, 113
- pointer-integer conversion 198–199, 205
- pointers and subscripts 97, 99, 217
- pointers, array of 107
- pointers, operations permitted on 103
- Polish notation 74
- pop function 77
- portability 3, 37, 43, 49, 147, 151, 153, 185
- position of braces 10
- postfix ++ and -- 46, 105
- pow library function 24, 251
- power function 25, 27
- #pragma 233
- precedence of operators 17, 52, 95, 131–132, 200
- prefix ++ and -- 46, 106
- preprocessor, macro 88, 228–233
- preprocessor name, __FILE__ 254
- preprocessor name, __LINE__ 254
- preprocessor names, predefined 233
- preprocessor operator, # 90, 230
- preprocessor operator, ## 90, 230
- preprocessor operator, defined 91, 232
- primary expression 200
- printf function 87
- printf conversions, table of 154, 244
- printf examples, table of 13, 154
- printf library function 7, 11, 18, 153, 244
- printing character 249
- program arguments *see* command-line arguments
- program, calculator 72, 74, 76, 158
- program, cat 160, 162–163
- program, character count 18
- program, dcl 125
- program, echo 115–116
- program, file concatenation 160
- program, file copy 16–17, 171, 173
- program format 10, 19, 23, 40, 138, 191
- program, fsize 181
- program, keyword count 133
- program, line count 19
- program, list directory 179
- program, longest-line 29, 32
- program, lower case conversion 153
- program, pattern finding 67, 69, 116–117
- program readability 10, 51, 64, 86, 147
- program, sorting 108, 119
- program, table lookup 143
- program, temperature conversion 8–9, 12–13, 15
- program, undcl 126
- program, white space count 22, 59
- program, word count 20, 139
- promotion, argument 45, 202
- promotion, integral 44, 197
- prototype, function 26, 30, 45, 72, 120, 202
- ptinrect function 130
- ptrdiff_t type name 103, 147, 206
- push function 77
- pushback, input 78
- putc library function 161, 247
- putc macro 176
- putchar library function 15, 152, 161, 247
- puts library function 164, 247
- qsort function 87, 110, 120
- qsort library function 253
- qualifier, type 208, 211
- quicksort 87, 110
- quote character, ' 19, 37–38, 193
- quote character, " 8, 20, 38, 194
- \r carriage return character 38, 193
- raise library function 255
- rand function 46
- rand library function 252
- RAND_MAX 252
- read system call 170
- readdir function 184
- readlines function 109
- realloc library function 252
- recursion 86, 139, 141, 182, 202, 269
- recursive-descent parser 123
- redirection *see* input/output redirection
- register, address of 210
- register storage class specifier 83, 210
- relational expression, numeric value of 42, 44
- relational operators 16, 41, 206
- removal of definition *see* #undef
- remove library function 242
- rename library function 242
- reservation of storage 210
- reserved words 36, 192
- return from main 26, 164
- return statement 25, 30, 70, 73, 225
- return, type conversion by 73, 225
- reverse function 62
- reverse Polish notation 74
- rewind library function 248
- Richards, M. 1
- right shift operator, >> 49, 206
- Ritchie, D. M. xi
- sbrk system call 187
- scaling in pointer arithmetic 103, 198
- scanf assignment suppression 157, 245
- scanf conversions, table of 158, 246
- scanf library function 96, 157, 246
- scientific notation 37, 73
- scope 195, 227–228
- scope, lexical 227
- scope of automatics 80, 228
- scope of externals 80, 228
- scope of label 66, 222, 228
- scope rules 80, 227
- screen output 15, 152, 163, 170
- SEEK_CUR, SEEK_END, SEEK_SET 248
- selection statement 223

- self-referential structure 140, 213
- semicolon 10, 15, 18, 55, 57
- separate compilation 67, 80, 227
- sequencing of statements 222
- setbuf library function 243
- setjmp library function 254
- <setjmp.h> header 254
- setvbuf library function 243
- Shell, D. L. 61
- shellsort function 62
- shift operators 48, 206
- short type 10, 36, 196, 211
- side effects 53, 90, 200, 202
- SIG_DFL, SIG_ERR, SIG_IGN 255
- sign extension 44–45, 177, 193
- signal library function 255
- <signal.h> header 255
- signed character 44, 195
- signed type 36, 211
- sin library function 251
- sinh library function 251
- size of numbers 9, 18, 36, 257
- size of structure 138, 204
- sizeof operator 91, 103, 135, 203–204, 247
- size_t type name 103, 135, 147, 204, 242
- sorting, lexicographic 118
- sorting, numeric 118
- sorting program 108, 119
- sorting text lines 107, 119
- specifier, auto storage class 210
- specifier, enum 39, 215
- specifier, extern storage class 31, 33, 80, 210
- specifier, missing storage class 211
- specifier, register storage class 83, 210
- specifier, static storage class 83, 210
- specifier, storage class 210
- specifier, struct 212
- specifier, type 211
- specifier, union 212
- splicing, line 229
- sprintf library function 155, 245
- sqrt library function 251
- squeeze function 47
- srand function 46
- srand library function 252
- sscanf library function 246
- standard error 161, 170
- standard headers, table of 241
- standard input 151, 161, 170
- standard output 152, 161, 170
- stat structure 180
- stat system call 180
- statement terminator 10, 55
- statements 222–225
- statements, sequencing of 222
- stat.h include file 180–181
- static function declaration 83
- static storage class 31, 83, 195
- static storage class specifier 83, 210
- static variables, external 83
- static variables, internal 83
- statics, initialization of 40, 85, 219
- <stdarg.h> header 155, 174, 254
- <stddef.h> header 103, 135, 241
- stderr 161, 163, 242
- stdin 161, 242
- <stdio.h> contents 176
- <stdio.h> header 6, 16, 89–90, 102, 151–152, 241
- <stdlib.h> header 71, 142, 251
- stdout 161, 242
- storage allocator 142, 185–189
- storage class 195
- storage class, automatic 31, 195
- storage class declaration 210
- storage class specifier 210
- storage class specifier, auto 210
- storage class specifier, extern 31, 33, 80, 210
- storage class specifier, missing 211
- storage class specifier, register 83, 210
- storage class specifier, static 83, 210
- storage class, static 31, 83, 195
- storage, definition of 210
- storage order of array 112, 217
- storage, reservation of 210
- strcat function 48
- strcat library function 249
- strchr library function 249
- strcmp function 106
- strcmp library function 249
- strcpy function 105–106
- strcpy library function 249
- strcspn library function 250
- stream, binary 160, 241–242
- stream, text 15, 151, 241
- strerror library function 250
- strftime library function 256
- strindex function 69
- string concatenation 38, 90, 194
- string constant 7, 20, 30, 38, 99, 104, 194
- string constant, initialization by 86, 219
- string constant, wide 194
- string, length of 30, 38, 104
- string literal *see* string constant
- string, type of 200
- <string.h> header 39, 106, 249
- strlen function 39, 99, 103
- strlen library function 250
- strncat library function 249
- strncmp library function 249
- strncpy library function 249
- strpbrk library function 250
- strchr library function 249
- strspn library function 250
- strstr library function 250
- strtod library function 251
- strtok library function 250
- strtol, strtoul library functions 252
- struct specifier 212
- structure arrays, initialization of 133
- structure declaration 128, 212
- structure initialization 128, 219
- structure member name 128, 213
- structure member operator, . 128, 201
- structure, nested 129
- structure pointer operator, -> 131, 201

- structure, pointer to 136
 - structure reference semantics 202
 - structure reference syntax 202
 - structure, self-referential 140, 213
 - structure, size of 138, 204
 - structure tag 128, 212
 - structures, arrays of 132
 - structures, mutually recursive 140, 213
 - subarray argument 100
 - subscripting, array 22, 97, 201, 217
 - subscripts and pointers 97, 99, 217
 - subscripts, negative 100
 - subtraction operator, - 41, 205
 - subtraction, pointer 103, 138, 198
 - suffix, constant 193
 - swap function 88, 96, 110, 121
 - switch statement 58, 75, 223
 - symbolic constants, length of 35
 - syntax notation 194
 - syntax of variable names 35, 192
 - syscalls.h include file 171
 - system calls 169
 - system library function 167, 253
-
- \t tab character 8, 11, 38, 193
 - table lookup program 143
 - table of escape sequences 38, 193
 - table of operators 53
 - table of printf conversions 154, 244
 - table of printf examples 13, 154
 - table of scanf conversions 158, 246
 - table of standard headers 241
 - tag, enumeration 215
 - tag, structure 128, 212
 - tag, union 212
 - talloc function 142
 - tan library function 251
 - tanh library function 251
 - temperature conversion program 8–9, 12–13, 15
 - tentative definition 227
 - terminal input and output 15
 - termination, program 162, 164
 - text lines, sorting 107, 119
 - text stream 15, 151, 241
 - Thompson, K. L. 1
 - time library function 256
 - <time.h> header 255
 - time_t type name 255
 - tmpfile library function 243
 - TMP_MAX 243
 - tmpnam library function 243
 - token 191, 229
 - token concatenation 90, 230
 - token replacement 229
 - tolower library function 153, 166, 249
 - toupper library function 166, 249
 - translation, order of 228
 - translation phases 191, 228
 - translation unit 191, 225, 227
 - tree, binary 139
 - tree, parse 123
 - treeprint function 142
 - trigraph sequence 229
 - trim function 65
 - truncation by division 10, 41, 205
 - truncation of floating point 45, 197
 - two-dimensional array 110, 112, 220
 - two-dimensional array, initialization of 112, 220
 - type conversion by return 73, 225
 - type conversion operator *see* cast
 - type conversion rules 42, 44, 198
 - type declaration 216
 - type declaration, inconsistent 72
 - type equivalence 221
 - type, incomplete 212
 - type names 220
 - type of constant 37, 193
 - type of string 200
 - type qualifier 208, 211
 - type specifier 211
 - type specifier, missing 211
 - typedef declaration 146, 210, 221
 - types, arithmetic 196
 - types, derived 1, 10, 196
 - types, floating 196
 - types, fundamental 9, 36, 195
 - types, integral 196
 - types.h include file 181, 183
-
- ULONG_MAX 252
 - unary minus operator, - 203–204
 - unary plus operator, + 203–204
 - unbuffered getchar 171
 - unbuffered input 170
 - undcl program 126
 - #undef 90, 172, 230
 - underflow 41, 250, 255
 - underscore character, _ 35, 192, 241
 - ungetc library function 166, 247
 - ungetch function 79
 - union, alignment by 186
 - union declaration 147, 212
 - union initialization 219
 - union specifier 212
 - union tag 212
 - unions, operations on 148
 - UNIX file system 169, 179
 - unlink system call 174
 - unsigned char type 36, 171
 - unsigned character 44, 195
 - unsigned constant 37, 193
 - unsigned long constant 37, 193
 - unsigned type 36, 50, 196, 211
 - usual arithmetic conversions 42, 198
-
- \v vertical tab character 38, 193
 - va_list, va_start, va_arg, va_end 155, 174, 245, 254
 - variable 195
 - variable, address of 28, 94, 203
 - variable, automatic 31, 74, 195
 - variable, external 31, 73, 195
 - variable length argument list 155, 174, 202, 218, 225, 254

- variable names, length of 192
- variable names, syntax of 35, 192
- vertical tab character, `\v` 38, 193
- `void * pointer` 93, 103, 120, 199
- `void` argument list 33, 73, 218, 225
- `void` type 30, 196, 199, 211
- `volatile` qualifier 196, 211
- `vprintf`, `vfprintf`, `vsprintf` library functions 174, 245

- `wchar_t` type name 193
- `while` statement 10, 60, 224
- `while` vs. `for` 14, 60
- white space 191
- white space characters 157, 166, 245, 249
- white space count program 22, 59
- wide character constant 193
- wide string constant 194
- word count program 20, 139
- `write` system call 170
- `writelines` function 109

- `\x` hexadecimal escape sequence 37, 193

- zero, omitted test against 56, 105