

**A Book on C**

**Programming in C**

**Fourth Edition**

**AI Kelley / Ira Pohl**

University of Caljfornia

Santa Cruz

...

**TT**

ADDISON-WESLEY

*Boston* • *San Francisco* • *New York· Toronto* • *jIy[ontreal London* • *Munich* • *Paris* • *Madrid*

*Capetown* • *Sydney' Tokyo* • *Singapore* • *Mexico* Cit:J1



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks, Where those designatious appear in this book, and we were aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for special sales. For more informa tion, please contact:

Pearson Education Corporate Sales Division

201 W. 103rd Street

Indianapolis, IN 46290

(800) 428-5331

corpsales@pearsoned.com

Visit AW on the Web: www.awl.com/cseng/

*Librmy of Congress Cataloging-in-Publication Data*

Kelley,Al

A book on C: programming in C I Al Kelley, Ira Pohl. -4th ed.

p. cm.

Includes bibliographical references and index

ISBN 0-201-18399-4

1. C (Computer program language) 1. Poh!, Ira, II. Title.

QA76.73.C15K44 1997

**For our Parents**

005.13'3--dc21

Copyright © 1998 by Addison-Wesley

97-44551 CIP

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 other wise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Text printed on recycled and acid-free paper.

ISBN 0201183994

9 1011121314 MA 04030201

9th Printing November 2001

**Contents**

**Preface**

**Chapter 0**

**Starting from Zero**

0.1 Why C?

0.2 ANSI C Standard

0.3 From C to C++

0.4 From C and C++ to Java

**Chapter 1**

**An Overview of C**

1.1 Programming and Preparation

1.2 Program Output

1.3 Variables, Expressions, and Assignment 1.4 The Use of #defi ne and #i ncl ude 1.5 The Use of pri ntfO and scanfO 1.6 Flow of Control

1.7 Functions

Call-by-Value

1.8 Arrays, Strings, and Pointers

Arrays

Strings

Pointers

1.9 Files



**xvii**

1

2

3

3

4

5

5

6

10

1 3

18

21

29

35

36

37

39

42

47

viii ., Contents

1.10 Operating System Considerations

Writing and Running a C Program

Interrupting a Program

Typing an End-of-file Signal

Redirection of the Input and the Output

Summary

Exercises

**Chapter 2**

**lexical Elements, Operators, and the C System** 2.1 Characters and lexical Elements

2.2 Syntax Rules

2.3 Comments

2.4 Keywords

2.5 Identifiers

2.6 Constants

2.7 String Constants

2.8 Operators and Punctuators

2.9 Precedence and Associativity of Operators

2.10 Increment and Decrement Operators

2.11 Assignment Operators

2.12 An Example: Computing Powers of 2

2.13 The C System

The Preprocessor

The Standard library

Summary

Exercises

**Chapter 3**

**The Fundamental Data Types**

3.1 Declarations, Expressions, and Assignment 3.2 The Fundamental Data Types

3.3 Characters and the Data Type char

3.4 The Data Type i nt

3.5 The Integral Types short, long, and unsi gned 3.6 The Floating Types 3.7 The Use of typedef

3.8 The si zeof Operator

3.9 The Use of getcharO and putcharO

3.10 Mathematical Functions

The Use of abs 0 and fabs 0

UNIX and the Mathematics library

53

53

56

56

56

58

60

**69**

70

73

75

77

78

79

80

81

83

85

87

89

91

91

92

96

98

**107** 107

110

111

116

11 7

119

122

122

124

127

130

130

., Contents ix

3.11 Conversions and Casts 131 The Integral Promotions 131 The Usual Arithmetic Conversions 131 Casts 133

3.12 Hexadecimal and Octal Constants 134 e13 Summary 137 Exercises 138

**Chapter** 4

**Flow of Control 147** 4.1 Relational, Equality, and logical Operators 147 4.2 Relational Operators and Expressions 149 4.3 Equality Operators and Expressions 152 4.4 logical Operators and Expressions 154

Short-circuit Evaluation 157 4.5 The Compound Statement 157 4.6 The Expression and Empty Statement 158 4.7 The if and the i f-el se Statements 159 4.8 The whi 1 e Statement 163 4.9 The for Statement 167 4.10 An Example: Boolean Variables 169 4.11 The Comma Operator 171 4.12 The do Statement 172 4.13 An Example: Fibonacci Numbers 174 4.14 The goto Statement 178 4.15 The break and conti nue Statements 179 4.16 The swi tch Statement 181 4.17 The Conditional Operator 182

Summary 184 Exercises 185

**Chapter** 5

**Functions 197** 5.1 Function Definition 197 5.2 The return Statement 200 5.3 Function Prototypes 201

Function Prototypes in C++ 202 5.4 An Example: Creating a Table of Powers 203 5.5 Function Declarations from the Compiler's Viewpoint 204

limitations 205 5.6 An Alternate Style for Function Definition Order 206 5.7 Function Invocation and Call-by-Value 207

x " Contents

5.8 Developing a Large Program

What Constitutes a Large Program?

5.9 Using Assertions

209 212 212

" Contents

6.14 Arguments to ma; nO

6.15 Ragged Arrays

6.16 Functions as Arguments

xi

290

292

293

5.10 Scope Rules

Parallel and Nested Blocks

Using a Block for Debugging

5.11 Storage Classes

The Storage Class auto

The Storage Class extern

The Storage Class reg; ster

The Storage Class stat; c

5.12 Static External Variables

5.13 Default Initialization

5.14 Recursion

Efficiency Considerations

5.15 An Example: The Towers of Hanoi Summary

Exercises

**Chapter 6**

**Arrays, Pointers, and Strings**

6.1 One-dimensional Arrays

Initialization

Subscripting

213

215

216 216 216 217 219 220 221

223

223

227 228 233

235

**245** 245

246 247

Functions as Formal Parameters in Function Prototypes

6.17 An Example: Using Bisection to Find the Root of a Function The Kepler Equation

6.18 Arrays of Pointers to Function

6.19 The Type Qualifiers const and vol at; 1 e

Summary

Exercises

**Chapter 7**

**Bitwise Operators and Enumeration Types**

7.1 Bitwise Operators and Expressions

Bitwise Complement

Two's Complement

Bitwise Binary Logical Operators

Left and Right Shift Operators

7.2 Masks

7.3 Software Tools: Printing an ; nt Bitwise

7.4 Packing and Unpacking

Multibyte Character Constants

7.5 Enumeration Types

296 296 300 302 307 309 **311**

**331** 331

333 333 334 335 337 338 341

344 345

6.2 6.3 6.4 6.5 6.6 6.7 6.8

6.9

Pointers

Call-by-Reference

The Relationship Between Arrays and Pointers Pointer Arithmetic and Element Size

Arrays as Function Arguments

An Example: Bubble Sort

Dynamic Memory Allocation With call oc 0 and mall oc 0 Offsetting the Pointer

248 252 253 255 256 257 259 262

7.6 An Example: The Game of Paper, Rock, Scissors Summary

Exercises

**Chapter 8**

**The Preprocessor**

8.1 The Use of #i ncl ude

8.2 The Use of #def; ne

Syntactic Sugar

348 356 357

**365** 365 366 367

6.10 6.11 6.12

An Example: Merge and Merge Sort

Strings

String-Handling Functions in the Standard Library Multidimensional Arrays

Two-dimensional Arrays

The Storage Mapping Function

Formal Parameter Declarations

Three-dimensional Arrays

Initialization

The Use of typedef

263 270 272 277 278 279 279 280 281 282

8.3

8.4

8.5

8.6

8.7

8.8

8.9

8.10 8.11 8.12

Macros with Arguments

The Type Definitions and Macros in *stddef.h* An Example: Sorting with qsortO An Example: Macros with Arguments The Macros in *stdio.h* and *ctype.h* Conditional Compilation

The Predefined Macros

The Operators # and ##

The assertO Macro

The Use of #error and #pragma

368 371 372 377 382 384 387 387 388 389

6.13 Arrays of Pointers

284

8.13

Line Numbers

390

xii ., Contents

8.14 Corresponding Functions

8.1 5 An Example: Quicksort

Summary

Exercises

**Chapter 9**

**Structures and Unions**

9.1 Structures

9.2 Accessing Members of a Structure

9.3 Operator Precedence and Associativity: A Final Look 9.4 Using Structures with Functions

9.5 Initialization of Structures

9.6 An Example: Playing Poker

9.7 Unions

9.8 Bit Fields

9.9 An Example: Accessing Bits and Bytes

9.10 The ADT Stack

Summary

Exercises

**Chapter 10**

**Structures and List Processing**

10.1 Self-referential Structures

10.2 Linear Linked Lists

Storage Allocation

10.3 List Operations

10.4 Some List Processing Functions

Insertion

Deletion

10.5 Stacks 

10.6 An Example: Polish Notation and Stack Evaluation 10.7 Queues

10.8 Binary Trees

Binary Tree Traversal

Creating Trees

10.9 General Linked Lists

Traversal 

The Use of callocO and Building Trees

Summary

Exercises

390

391

394

396

**407** 407

411

415

416

418

419

424

427

429

430

435

437

**447 **447

449 

450

451

455

458

459

460

464

471

475

477

478

479

482

482

484

485



., Contents xiii

**Chapter 11**

**Input/Output and the Operating System 493** 11.1 The Output Function pri ntfO 493 11.2 The Input Function scanfO 499 11.3 The Functions fpri ntfO, fscanfO, spri ntfO,

and sscanfO 503 11.4 The Functions fopenO and fcloseO 505 11. 5 An Example: Double Spacing a File 507 11.6 Using Temporary Files and Graceful Functions 510 11.7 Accessing a File Randomly 513 11.8 File Descriptor Input/Output 514 11.9 File Access Permissions 517 11.10 Executing Commands from Within a C Program 518 11. 11 Using Pipes from Within a C Program 520 11.12 Environment Variables 521 11.13 The C Compiler 522 11.14 Using the Profiler 524 11.15 Libraries 526 11.16 How to Time C Code 528 11.17 The Use of make 532 11.18 The Use of *touch* 538 11.19 Other Useful Tools 539

Summary 541 Exercises 542

**Chapter 12**

**Advanced Applications 555** 12.1 Creating a Concurrent Process with forkO 555 12.2 Overlaying a Process: the exec ... 0 Family 558

Using the spawn ... 0 Family 560 12.3 Interprocess Communication Using pi peO 561 12.4 Signals 564 12.5 An Example: The Dining Philosophers 567 12.6 Dynamic Allocation of Matrices 571

Why Arrays of Arrays Are Inadequate 571 Building Matrices with Arrays of Pointers 572 Adjusting the Subscript Range 575 Allocating All the Memory at Once 577

12.7 Returning the Status 579 Summary 585 Exercises 586

xiv ... Contents 

**Chapter 13**

**Moving from C to C++**

13.1 Output

13.2 Input

13.3 Functions

13.4 Classes and Abstract Data Types

13.5 Overloading

13.6 Constructors and Destructors

13.7 Object-oriented Programming and Inheritance 13.8 Polymorphism

13.9 Templates

13.10 C++ Exceptions

13.11 Benefits of Object-oriented Programming Summary

ExerCises

**Chapter 14**

**Moving from C to Java**

14.1 Output

14.2 Variables and Types

14.3 Classes and Abstract Data Types 

14.4 Overloading

14.5 Construction and Destruction of Class Types 14.6 Object-oriented Programming and Inheritance 14.7 Polymorphism and Overriding Methods

14.8 Applets

14.9 Java Exceptions

14.1 0 Benefits of Java and OOP

Summary

Exercises

**Appendix A**

**The Standard Library**

A.1 Diagnostics: <assert. h>

A.2 Character Handling: <ctype. h>

Testing a Character 

Mapping a Character

A.3 Errors: <errno.h>

A.4 Floating Limits: <float. h>

A.S Integral Limits: <limits.h>

A.6 Localization: <locale.h>

A. 7 Mathematics: <math. h>

**593**

594

595

599

601

603

606

608

610

612

614

615

617

619

**625**

626

627

629

631

631

632

633

635

636

638

639

640

**641**

641

642

642

643

643

644

645

645

646

... Contents xv

A.8 NonlocalJumps: <setjmp. h> 649 A.9 Signal Handling: <signal. h> 650 A.10 Variable Arguments: <stdarg. h> 651 A.ll Common Definitions: <stddef. h> 652 A.12 Input/Output: <stdi o. h> ... . 653

Opening, Closing, and. ~ondltl?nlng a File 655 Accessing the File Position Indicator 656 Error Handling 658 Character Input/Output 658 Formatted Input/Output 660 Direct Input/Output 662 Removing or Renaming a File 662

A.13 General Utilities: <stdl i b. h> 663 Dynamic Allocation of Memory 663 Searching and Sorting 664 Pseudo Random-Number Generator 665 Communicating with the Environment 665 Integer Arithmetic 666 String Conversion 666 Multibyte Character Functions 668 Multibyte String Functions 669 Leaving the Program . 670 

A.14 Memory and String Handling: <stn ng. h> 670 Memory-Handling Functions 671 String-Handling Functions 671

A.15 Date and Time: <time.h> 675 Accessing the Clock 676 Accessing the Time 676

A.16 Miscellaneous 680 File Access 680 Using File Descriptors 681 Creating a Concurrent Process 681 Overlaying a Process 682 Interprocess Communication 683 Suspending Program Execution 683 **Appendix B** 

**Language Syntax 685** B.l Program 685 B.2 Function Definition 686 B.3 Declaration 686 B.4 Statement 688 B.5 Expression 689

x~ • Con~n~

B.6 B.7 B.8

Constant

String literal Preprocessor

690 691

**Preface **

**Appendix C**

**ANSI C Compared to Traditional C** e.l Types

e.2 Constants

e.3 Declarations

CA Initializations

e.5 ExpresSions

e.6 Functions

e.7 Conversions

e.8 Array Pointers

e.9 Structures and Unions

e.l0 Preprocessor

e.l1 Header Files

e. 12 Miscellaneous

**Appendix 0**

**ASCII Character Codes**

**Appendix E**

**Operator Precedence and Associativity Index**

692

**693**

693

694

695

695

696

696

698

698

699

700

701

701

**703**

**705**

**707**

A Book on C conveys an appreciation for both the elegant simplicity and the power of this general-purpose programming language. By presenting interactive running pro grams from many application areas, this book describes the ANSI version of the C lan guage. The complete language is presented in a step-by-step manner, along with many complete working programs.

Where appropriate, we discuss the differences between traditional C and ANSI e. (Traditional C still remains in wide use.) Dozens of example programs are available to illustrate each important language feature, and many tables summarize key informa tion and provide easy access for later reference. Each chapter ends with a summary and exercises. The summary reviews key elements presented in the chapter, and the exer cises augment and extend the text.

This book assumes a general-purpose knowledge of the C language. It is intended for use in either a first or second programming course. However, it can be readily used in conjunction with courses on topics such as comparative programming languages, com putationallinguistics, data structures, database systems, fractal geometry, graphicS, numerical analysis, operating systems, programming methodology, and scientific appli cations. C is suitable for applications from each of these domains, and all features of C needed to code such applications are explained. This book is appropriate for a data structures course because advanced data structuring features such as enumeration types, unions, self-referential structures, and ragged arrays are discussed. For operat ing systems courses concerned with UNIX or Windows 95/NT, the book explores the file structure and systems routines that enable the C programmer to add to existing sys tems libraries and understand the C code underlying the operating system. For applica tions programming and scientific programming, there is discussion of how to write sample function libraries. Statistics, root finding, sorting, text manipulation, file han dling, and game playing are all represented with working code.

**7**

xviii l' Preface

*New Java Section.* In Chapter 14, "Moving from C to Java," we discuss how the C pro grammer can very naturally and easily begin programming in Java, a language of inter est for work on the Internet. The Java programming language borrows ideas from both C and c++ and is designed to run in a machine- and system-independent manner. This makes it suitable for Internet work, such as writing applets for Web pages that get used by browsers. Because Java is an extension of C and C++, it is readily learned by the C programmer.

*Complete ANSI* C *Language.* Computer profesSionals will have access to a complete treatment of the language, including enumeration types, list processing, and the operat ing system interface. Chapter 1, "An Overview of *C,"* presents an overview of the lan guage. After reading this chapter, the professional will already be able to write C code. Since the chapters are self-contained, the knowledgeable reader can skip to particular sections as needed. Chapter 11, "Input/Output and the Operating System," gives a thor ough introduction to the connections to the operating system. This information will benefit the professional systems programmer needing to use C to work 'vi thin an MS- DOS or UNIX environment.

*Interactive Environment.* This book is written entirely with the modern interactive environment in mind. Experimentation is encouraged throughout. Keyboard and screen input/output is taken as the norm, and its attendant concerns are explained. Thus, the book is appropriate for users of small home and business computers as well as to users of large interactive systems. We assume that the reader will have access to an interac

tive ANSI C system. During the writing of this book, we used a number of different C systems: various Borland and Microsoft compilers running on IBM-compatible Pentium machines, the GNU *gee* compiler and native compilers running on various workstations from DEC, SGI, and Sun, and the C compiler that runs on the Cray supercomputer in San Diego.

*Working Code.* Our approach to describing the language is to use examples, explana tion, and syntax. Working code is employed throughout. Small but useful examples are provided to describe important technical points. Small because small is comprehensi

ble. Useful because programming is based on a hierarchy of building blocks and ulti mately is pragmatic. The programs and functions deSCribed in the book can be used in actual systems. The authors' philosophy is that one should experiment and enjoy.

*Dissections.* We use highlighted "dissections" on many programs and functions throughout the book Dissection is a unique pedagogical tool first developed by the authors in 1984 to illUminate key features of working code. A dissection is similar to a structured walk-through of the code. Its jntention is to explain to the reader newly encountered programming elements and idioms found in working code.

'f Preface xix

.. . b k is constructed to be very flexible in its use. Chapte: I, *Flexible* Orgamzatl~~. :his 00 rts The first part explains the crucial programmmg "An Overview of C, Is:n two ~a " / t ut material that must be understood by techniques needed for mteractIve I~~~~ ~~ s~rv~y the entire language and will be com alL The second part ?f Chapter 1 goe. f 'liar with comparable features from other prehensible to expenenced programmers amJd , a first programming course. *Caution:*

s This second part can be postpone 111

language . ld ost one the second part of Chapter 1. Beginning programmers shou p p . d the C S stem" describes the lexical level Chapter 2, "Lexical Eleme~ts, ~perat~~~h :~e SeleCti~elY e~ployed to illustrate C lan of the language and sy.ntact!c ru es, ~decide to teach Backus-Naur-Form (BNF) notation guage constructs. The mstructor ma\ 't 'thout any loss of continuity. The book uses as described in Chapter 2 ,or ,may onn 1 t v~e student can learn this standard form of BNF-style syntactic descnptl?n~ so :ha dd'tion language components are thoroughly programming language descnptIOn. n a I, ' described by example and ordinary explanatIOn.

. . d si ned to be a valuable reference to the C language. *Reference Work.* ThIS book IS e ~oncisell illustrate key areas of the language. The Throughout the book, many tables 1 wit~ its associated header files, is described in complete AN.S1 C s,;anda~d h~ar~, ti~~:ry." Sections in the appendix are devoted to the AppendIX A, The Stan ar eader files such as *etype.h, stdio.h,* and *string.h.* Where explaining each of the stand~~d ~ to illustrate the use of a particular construct or appropriate, example code IS gIven  function. "'de the complete syntax of the C language . ..:l~" B "L nguage Syntax we proVl . In AppenLUll. , a 'd. T d't' I C " we list the major dIfferences . C "ANSI C Compare to ra 1 IOna , . d  ~t~~~~N~I C and traditional C. Finally, special care has been taken to make the m ex easy to use and suitable for a reference work.

hers 3 through 10 cover the C language feature *The Complete ANSI* C Lan9ua~e. C a~~Scussed that may be omitted on first reading without loss of comprehensIOn, I hSO, esrre. be omI'tted' in a first course. Machine- h I ge and t elf use can \_. tively new to t e angua, . 'd tions and floating-point representatIOn d l' atures such as word SIze consl era

by feature. Many advanced ~oPI7 ar~ . d For example enumeration types are rela

are depen emp ehnt aSIze . e d but many of the details need not concern the beginner. , sor, WhIch IS use to ex elf a function call. Their use can reduce

" Processor" is devoted entirely to the preproces- *The Preprocessor.* Chapter 8, The rep d t 't'on of the C language. Macros can be ,. d t nd the power an no a 1

used to generat~ inli~e code that takes t~:s~n~~eaOdetailed discussion of the preproces program executIOn tlme. The chap~er ~ ANSI committee. In traditional C, the prepro sor, including ne",: features added y e iler to another. In ANSI C, the functionality of cessor varies conSIderably from one comp , .

the preprocessor has been completely speCIfIed.

xx l' Preface

*Recursion and List Processing.* Chapter S, "Functions," has a careful discuSSion of recursion, which is often a mystifying topic for the beginner. The Use of recursion is illustrated again in Chapter 8, "The Preprocessor," With the quicksort algOrithm and in Chapter 10, "Structures and List ProCessing," With basic list prOceSSing techniques. A ming thorough knowledge of *list* processing techniques is necessary in advanced program. and data structure courses.

l' Preface xxi

**Acknowledgments** the chief technical editor for to Debra Dolsberry, who acted as I er to create PostSCrIpt ~lles Our spedal thanks go I responsible for using FrameMa {kS also go to Robert FIeld,

this book She was large Yn of this book. Our speetal than

chief technical reviewer

*Operating System Connection.* Chapter II, "Input/Ou tpu t and the Opera ting Sys tern," makes the operating system conneCtion. In this chapter, we explain how to do file pro. cessing and discuss at length the various input/output functions in the standard

library. We also explain how to execute a system command from Within a C program and how to set file permissions and USe of environment variables. We give explicit examples shoWing the Use of the proftler, the librarian, and the *make* facility.

*Advanced ApPlications.* We discuss a number of advanced applications in Chapter 12, "Advanced Applications." We present topics such as creating concurrent processes, overlaying a process, interprocess communication, and Signals, along With Working COde. Also, We discuss the dYllamic allocation of vectors and matrices for eng;neers and

sdentists. These advanced tOPles can be used selectively according to the needs of the audience. They could form the basis for an excellent second course in programming

Psuitable for the typesettt i~ View California, who acted as t d su gestions extremely 

he

arcPlace Systems, d Moufntht~s boole' We found his expertise a~ns "gand the "Dining Phi- .,. n 0 I. 5 "Funcno , d for the first he 1 10 f H' 01'" picture in Chapter , . g" are due to John e "T wer 0 an d List Processm , valuable. Teo . Chapter 10, "Structures an. h nks go to him, too. losophers" Plctur~ ~l1fornia, Riverside. Our speml t ~ 'th helpful suggestions: Mur. Pillis, UruverSlty 0 k other people who prOVIded ~s WI hae! Beeson, San Jose State

We also want to th~:rsity of California, Santa Cruz, 1~;~dO State University, Ft. Col ray Baumgarten, Urn C ]"f rnia' Randolph Bentson, Co B ie Hewlett-Packard Co.: University, San Jose: a '.,0 of California, Berkeley; JOhn.;w ofuY Budd, University of lins; Jim Bloom, Umve.'" Yil of California, Santa Bru:bara, 1Ul ta Cruz; Jim Chrislock, Arizona, Tucson; Nl::d Jniversity of California, Santa *ie':* G1ntenbein, University of 

inc.; Skona Bnttam, umv~; ~yne, University of Cabforma, szan effDonnelly, UniversIty Minderaft, inc.; AI Co. I F 'tz AT&T Bell Laboratones, '. L onard Garrett, Temple

I

Wyoming, . e' Harry a , St te UrnversI, ., l'f' Lararm , h' . William Giles, Sao Jose a nl<amer University of Ca lor

practlee. lbis book can be used, too, as an aUXiliary text in adVanced cOlllPuter sdence courses that employ C as their implementation language.

of IllinOis, Urbana; DIC <: rG

"v'er SRl International, GeOr~la, ,ety San Jose California;

*TabIes, Summaries, and exercises.* Throughout Ihe book are many tables and lists that

University, Plnladdp l~~ of California, Berkeley; Jorge ~af 'a' Mike Johnson, Oregon Susan Graham, Umversl H 0 Auspex, inc., San Jose, CLOT" 'dro California; Carole

SUCCinctly sununanze *key* ideas. These tables aid and test language comprehension. For

nia, Santa Cruz; RObertlrsaxK~ith Jolly, Chabot College, san

e:,s St~te University; Dar.

example, C is *very* rich in operators and allows almost any useful combination of Opel'. ator mix. It is eSSential to understand order of evaluation and association of each of these operators separately and in combination. These points are llIustrated in tables throughout the text. As a reference tool, the tables and code ill'e easily lOoked up.

The exerdses test elementary features of the lilllguage and discuss advanced and sys. tem'dependent features. Mony exercisos are oriented to problem solVing, others test the reader's SYlltactic or Semantic understanding of C. Some exerdses include a tutorial diSCUSSion that is tangential to the text but may be of special interest to certain readers.

able leexercises to the audience. offer the instructor all leVels of question, so as to allow aSSignments sUit

State University, Corva 1 , tos California; Clifford Layton, og II University of Califor

R

Kelley, Cabrillo COlleg~, ~ifO;nia, Santa Cruz; Ch~~e M~~o;~uider: Geoffrey Pullum: rell Long, Uruverstty 0 Pleszkun, University of Co ora 'Santa Cruz Operation, Inc., nia, Santa Cruz; Anmew ta Cruz; Peter Rosenerantz, The sit of California, Santa University of Caltf~ruaie~:;ackard Co., Inc.; Peter sco.~' ~:::::r uJversity of Califonna, Mike Schoonover, ~w. f Washington, Seattle; Tl y , Sh Urnverslty 0 h' Cruz; Alan aw, n University of Denver. . Carter Shanklin for IS Santa Cruz; Matt Sta~~li;{C to thank our sponsoring I~d~or to thank John Fuller for In addition, we wou d ncouragement; and we wou <:e

enthusiasm, sup~ort, a~ e roduction of this book on C. his careful attentIOn to t e p

AI Kelley

g~~~~ity of California, Santa Cruz

**t **

****

**Chapter 0** 

**Starting from Zero**

Zero is the natural starting point in the C programming language. C counts from O. C uses 0 to mean false and not 0 to mean true. C array subscripts have 0 as a lower bound. C strings use 0 as an end-of-string sentinel. C pointers use 0 to designate a null value. C external and static variables are initialized to 0 by default. This book explains these ideas and initiates you into the pleasures of programming in C.

C is a general-purpose programming language that was originally designed by Dennis Ritchie of Bell Laboratories and implemented there on a PDP-ll in 1972. It was first used as the systems language for the UNIX operating system. Ken Thompson, the devel oper of UNIX, had been using both an assembler and a language named B to produce initial versions of UNIX in 1970. C was invented to overcome the limitations of B.

B was a programming language based on BCPL, a language developed by Martin Rich ards in 1967 as a typeless systems programming language. Its basic data type was the machine word, and it made heavy use of pointers and address arithmetic. This is con trary to the spirit of structured programming, which is characterized by the use of strongly typed languages, such as the ALGOL-like languages. C evolved from Band BCPL, and it incorporated typing.

By the early 1980s, the original C language had evolved into what is now known as *traditional* C by adding the vo; d type, enumeration types, and some other improve ments. In the late 1980s, the American National Standards Institute (ANSI) Committee X3]11 created draft standards for what is known as *ANSI* C or *standard* C. The commit tee added the vo; d i< type, function prototypes, a new function definition syntax, and more functionality for the preprocessor, and in general made the language definition more precise. Today. ANSI C is a mature, general-purpose language that is widely avail able on many machines and in many operating systems. It is one of the chief industrial programming languages of the world, and it is commonly found in colleges and univer sities everywhere. Also, ANSI C is the foundation for C++, a programming language that incorporates object-oriented constructs. This book describes the ANSI version of the C language, along with some topics in C++ and Java.

2 Chapter 0 'f Starting from Zero

**0.1 Why C?**

C is a small language. And small is beautiful in programming. C has fewer keywords than Pascal, where they are known as reserved words, yet it is arguably the more power fullanguage. C gets its power by carefully including the right control structures and data types and allowing their uses to be nearly unrestricted where meaningfully used.

The language is readily learned as a consequence of its functional minimality. Cis th: native language of UNIX, and UNIX is a major interactive operating system on workstatIOns, servers, and mainframes. Also, C is the standard development language for personal computers. Much of MS-DOS and OS/2 is written in C. Many windowing packa~es, d.atabase programs, graphics libraries, and other large-application packages are wntten m C.

. C is portable. Code written on one machine can be easily moved to another. C pro VIdes .the programmer with a standard library of functions that work the same on all machmes. Also, C has a built-in preprocessor that helps the programmer isolate any system-dependent code.

C is terse. C has a very powerful set of operators, and some of these operators allow t~e programmer to access the machine at the bit level. The increment operator ++ has a dIrect a~alo~ue in machine language on many machines, making this an efficient opera to:. I~dlrectIOn and address arithmetic can be combined within expressions to accom plIsh m one statement or expression what would require many statements in another !anguag~. For many programmers this is both elegant and efficient. Software productiv- :ty studIes show that programmers produce, on average, only a small amount of work mg cOd.e .each day. A language that is terse explicitly magnifies the underlying productIVIty of the programmer.

C is modular. C supports one style of routine, the external function, for which argu ments are passed call-by-value. The nesting of functions is not allowed. A limited form of priv~cy is provide.d by using the storage class static within files. These features, along WIth tools prOVIded by the operating system, readily support user-defined librar ies of functions and modular programming.

C ~s the baSis for c++ and Java. This means that many of the constructs and method ologIes that are routinely used by the C programmer are also used by the c++ and Java programmer. Thus, learning C can be considered a first step in learning c++ or Java.

C .i~. efficien: on most machines. Because certain constructs in the language are expllCltly machine-dependent, C can be implemented in a manner that is natural with respe~t to the machine's architecture. Because a machine can do what comes naturally, compiled C code can be very efficient. Of course, the programmer must be aware of any code that is machine-dependent.

0.2 'f ANSI C Standard 3

C is not without criticism. It has a complicated syntax. It has no automatic array ",V'CHUA" checldng. It makes multiple use of such symbols as 1, and ==. For example, a common programming error is to use the operator in place of the operator ==. Never theless, C is an elegant language. It places no straitjacket on the programmer's access to the machine. Its imperfections are easier to live with than a perfected restrictiveness.

C is appealing because of its powerful operators and its unfettered nature. A C pro grammer strives for functional modularity and effective minimalism. A C programmer welcomes experimentation and interaction. Indeed, experimentation and interaction are the hallmarks of this book

**0.2 ANSI C Standard**

The acronym ANSI stands for "American National Standards Institute." This institute is involved in setting standards for many!dnds of systems, including programming lan guages. In particular, ANSI Committee X3Jl1 is responsible for setting the standard for the programming language C. In the late 1980s, the committee created draft standards for what is known as ANSI C or standard C. By 1990, the committee had finished its work, and the International Organization for Standardization (ISO) approved the stan dard for ANSI C as well. Thus, ANSI C, or ANSI/ISO C, is an internationally recognized standard.

The standard specifies the form of programs written in C and establishes how these programs are to be interpreted. The purpose of the standard is to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of machines. Almost all C compilers now follow the ANSI C standard.

**0.3 From C to C++**

Today, C is widely available on PCs, workstations, and mainframes throughout the world. At the same time, machines and operating systems continue to evolve. To expand the C language or to restrain the use of its constructs in order to conform to a particular discipline is not in the spirit of C.

Although the C language itself is not being expanded, it often serves as the kernel for more advanced or more specialized languages. Concurrent C extends the language by

4 Chapter 0 .,. Starting from Zero

incorpOrating concurrency primi .

Small talk style objects Oth f tlves. Objective C extends the language b . . ta!; of ~ifferent forms 'of pa~:l1~~:.Of C are used on supercomputers to ia~~oav~~~:~ ost Important is C++ an ob' . Because it is an exten" Ject-onented language alread' . ware projects C++ . slO.n ~f C, it al10ws both C and C++ code t b m widespread use. from C to C++' ") IS readIly learned by the C programmer (50 Cehused on large soft- . . . ee apter 13 "M . , ovmg

**0.4 From C and C++ to Java**

Java was designed for w

and portabl . ark on the Internet. It allows th

machine T~;~og~ams that can be downloaded from the ~~rogrammer to write secure deSigned to rune~n p:~~a~ming language borrows ideas fr~~e~~~ ~un ~n your local defined in terms of a virtuCalme - a~d system-independent manner Its:n C++. and is  ~~~s~::I~~~~diverse Sys:~~~~~:~c~i:~:~ ~~a;~:~ is inhe~'e~tly ~~:~~~s a~~ J' ng on workstations. mg on a PC and various fla ava IS often used to write applets on

~~nd~~:, ~:~,:~e;e:~~~~hicaI ~se~ int::!~~~et~~~~::.tB~sC:~~: i~riowsers. Typ~cal1y, C to Java.") earne by the C programmer (5 Ch s an extensIOn of . ee apter 14, "MOving from



**Chapter 1** 

**An Overview of C**

This chapter gives an overview of the C programming language. A series of programs is presented, and the elements of each program are carefully explained. Experimentation and interaction are emphasized throughout the text. In this chapter, we emphasize how to use the basic input/output functions of C. Note carefully that all our C code also serves as C++ code and that all the ideas we discuss hold for C++ as well as for C. Of course, the C++ programmer has available a richer set of tools and techniques out of which programs can be constructed. (See Chapter 13, "Moving from C to CH.")

Except for Section 1.8, "Arrays, Strings, and Pointers," on page 36, everyone should read all the material in this chapter. Any reader who has had experience with arrays, pointers, and files in some other language can read all the sections of this chapter to get a more complete overview of C. Others can come back to the material when they feel they are ready. Everyone should read this chapter with the understanding that tech

nical details and further explanations will come in later chapters.

**1.1 Programming and Preparation**

Resident on the machine is a collection of special programs called the *operating system.* Commonly available operating systems include MS-DOS, OS/2, and UNIX. An operating system manages machine resources, provides software for the user, and acts as an interface between the user and the hardware. Among the many software packages pro

vided by the operating system are the C compiler and various text editors. The principal text editor on the UNIX system is called vi. Some systems, such as Borland C++, inte grate the text editor and the compiler. We assume that the reader is able to use some text editor to create files containing C code. Such files are called source files, and they

6 Chapter 1 T An Overview of C

are compiled on most UNIX systems with the cc command, which invokes the C com piler. Because the cc command invokes the compiler, the name of the command is also the name of the compiler. Thus, C compiler and cc compiler are used interchangeably. Roughly speaking, a compiler translates source code to object code that is executable. On UND( systems, this compiled code is automatically created in a file named *a.out.* On MS-DOS systems, this compiled code is automatically created in a file with the same na~e as the .c file, but with the .exe extension replacing the .c extension. At the end of thIS cha.pter, i~ Section 1.10, "Operating System Considerations," on page 53, we present m detaIl the steps necessary to edit, compile, and execute a program.

**1.2 Program Output**

Programs must communicate to be useful. Our first example is a program that prints on the screen the phrase "from sea to shining c." The complete program is

In file sea.c

#include <stdio.h>

i nt mai n(voi d)

{

printf("from sea to shining C\n");

1.2 T Program Output 7

• Dissection of the *sea* Program  

• #include <stdio.h>

A preprocessor is built into the C compiler. When the command to compile a program is given, the code is first preprocessed and then compiled. Lines that begin with a # communicate with the preprocessor. This #i ncl ude line causes the preprocessor to include a copy of the header file *stdio.h* at this point in the code. This header file is pro

vided by the C system. The angle brackets around <stdi o. h> indicate that the file is to be found in the *usual place,* which is system-dependent. We have included this file because it contains information about the pri ntfO function.

• int main(void) 

This is the first line of the function definition for mai nO. (We write parentheses after the name ma into remind the reader that main 0 is a function.) The two words i nt and vo; d are keywords, also called reserved words. They have special meaning to the com piler. In Section 2.4, "Keywords," on page 77, we will see that there are 32 keywords in C, including i nt and vo; d.

• int main(void)

{

}

return 0;

Every program has a function named main O. Program execution always starts with this function. The top line should be read as "main 0 is a function that takes no arguments

Using a text editor, we type this into a file whose name ends in .c. The chOice of a file name should be mnemonic. Let us suppose the program has been written in the file *sea.c.* To compile the program, we give the command

cc *sea.c*

If there are no errors in the code, the executable file *a.out* is created by this command. Now the command

*a.out*

executes the program and prints on the screen

from sea to shining C

and returns an i nt value." Here, the keyword i nt tells the compiler that this function returns a value of type i nt. The word i nt stands for *integer,* but the word *integer* itself cannot be used. The parentheses following ma; n indicate to the compiler that mai n is a function. This idea is confusing at first because what you see following main is (vo; d) , but only the parentheses 0 constitute an operator telling the compiler that ma; n is a function. The keyword voi d indicates to the compiler that this function takes no argu 

ments. When we write about functions such as main 0 and p ri ntf 0, we usually follow the name in print with parentheses. This indicates to the reader that we are discussing a function. (Many programming books follow this practice.)

.. {

Braces surround the body ofet function definition. They are also used to group state ments together.

8 Chapter 1 'f An Overview of C

III pri ntfO

The C system contains a standard library of functions that can be used in programs. This is a function from the library that prints on the screen. We included the header file sldio.h because it provides certain information to the compiler about the function printfO. (See exercise 14, on page 63.)

III "from sea to shining (\n"

A string constant in C is a series of characters surrounded by double quotes. This string is an argument to the function pri ntfO, and it controls what gets printed. The two characters \n at the end of the string (read backs lash n) represent a single character called newline. It is a nonprinting character. It advances the cursor on the screen to the beginning of the next line.

III printf("from sea to shining (\n")

This is a call to the pri ntfO function. In a program, the name of a function followed by parentheses causes the function to be called, or invoked. If appropriate, the paren theses may contain arguments. Here, when the pri ntfO function is invoked, it prints its argument, a string constant, on the screen.

III printf("from sea to shining (\n");

This is a statement. Many statements in C end vvith a semicolon.

III return 0;

This is a retu rn statement. It causes the value zero to be returned to the operating sys tem, which in turn may use the value in some way, but is not required to do so. (See Sec tion 12.7, "Returning the Status," on page 579, for further discussion.) Our use of this return statement keeps the compiler happy. If we do not use it, the compiler will com plain. (See exercise 4, on page 60.) One of the principal rnles of programming is "keep your compiler happy." 

III }

•

The right brace matches the left brace above, ending the function definition for main O .

1.2 v Program Output 9

The function pri ntfO acts to print continuously across the screen. It moves the cursor to the start of a new line when a newline character is read. The screen is a two-dimensional display that prints from left to right and top to bottom. To be read able, output must appear properly spaced on the screen.

We can rewrite our first program as follows:

#include <stdio.h>

i nt ma; n(voi d)

{ printf("from sea to ");

printf("shining C");

printfCII\n");

return 0;

}

Although it is different from the first version, it will produce the same o~tput. Each time pri ntfO is called, printing begins at the position w~ere the preVIOUS call, to pri ntfO left off. If we want to print our phrase on three hnes, we can use newlme

characters.

#include <stdio.h>

int mainCvoid)

{ pri ntf(,'from sea\n");

printf("to shining\n(\n");

return 0;

}

When executed, this program will print

from sea

to shining

(

10 Chapter 1" An Overview of C 

Let us write one additional variation on this program, one that will box the phrase in a rectangle of asterisks. It will show how each character, including blanks and newline characters, is Significant, and when it is executed, it will give some sense of the screen proportions.

In file sea2.c

#include <stdio.h>

11

1.3 " Variables, Expressions, and Assignment

In file marathon.c

*Ii'* The di stance Of a marathon in kilometers. \*1

#include <stdio.h>

int main(void)

{

float miles, yards;

int main(void) {

i nt

kilometers;

printf("\n\n\n\n\n\n\n\n\n\n");

p r i n t f ( II ,~ \* 'i<'1"h~ 'i"hh~ 'I, *i,* '{doh"", \* ,', j, '/d, \* \ nil) ;

pri ntf(" ,~ from sea i'\n") ;

pri ntf(" *i,* to shi ni ng C >"\n") ;

p ri ntf (" "'\*\*\*'~id'\*i"hhh~"''''\*i''~\*1d''~j(\n'') ;

printf("\n\n\n\n\n\n\n\n\n\n");

}

return 0;

}

miles 26;

yards 385; , kilometers = 1.609 R printf("\nA marathon return 0;

(miles + yards I *1760.0);*

is %f kilometers.\n\n", kilometers);

**1.3**

**Variables, ExpreSSions, and Assignment**

The output of the program is.

A marathon is 42.185970 kilometers. • Dissection of the *marathon *Program

We will write a program to convert the distance of a marathon in miles and yards to kilometers. In English units, a marathon is defined to be 26 miles and 385 yards. These numbers are integers. To convert miles to kilometers, we multiply by the conversion factor 1.609, a real number. In memory, computers represent integers differently from reals. To convert yards to miles, we divide by 1760.0, and, as we shall see, it is essential to represent this number as a real rather than as an integer.

Our conversion program will use variables capable of storing integer values and real values, In C, aU variables must be declared, or named, at the beginning of the program. A variable name, also called an identifier, consists of a sequence of letters, digits, and underscores, but may not start with a digit. Identifiers should be chosen to reflect their use in the program. In this way, they serve as documentation, making the program more readable.

,~ I • 1\* The distance of a marathon in kilometers. 

*Ii'* nd it *I* is a comment and is ignored by the Anyt~ing written betwe~nt~e ~~:~~~: star~ vvith a comment are listed in the index. compller. All programs m s ,

• int miles, yards;

. d ts end with a semicolon. ; nt is a key- This is a declaration. DeclaratIOns ~n ::a~~r:~~ language. It informs the compiler that word and is one of t~e f~ndamefnta ty?nt and are to take on integer values. Thus, the the variables followmg It are 0 type 1 .' variables mi 1 es and yards in this program are of type 1 nt.

• float kilometers; 

. . k word and is one of the fundamental types of the This is a de~la:atIOn~:~ ~~~~~l:r ~at the variables following it are of type f.loat and language. It m arms I Th the variable ki 1 ometers in this program IS of type are to take on real va ues. us, float.

12 Chapter 1 'V An Overview of C

II miles 26;

yards 385;

These are assignment statements. The equal sign is an assignment operator. The two numbers 26 and 385 are integer constants. The value 26 is assigned to the variable mi 1 es. The value 385 is assigned to the variable yards.

II kilometers 1.609 \* (miles + yards / 1760.0);

This is an assignment statement. The value of the expression on the right side of the equal sign is assigned to the variable ki 1 ometers. The operators ''<, +, and / stand for multiplication, addition, and division, respectively. Operations inside parentheses are performed first. Because division has higher precedence than addition, the value of the subexpression

yards / 1760.0

is calculated first. (See Appendix E, "Operator Precedence and Associativity.") That value is added to the value of the variable mi 1 es to produce a value that is then multi plied by 1. 609. This final value is then assigned to the variable ki lometers.

II printf("\nA marathon is %f kilometers.\n\n", kilometers);

This is a statement that invokes, or calls, the pri ntfO function. The function pri ntfO can have a variable number of arguments. The first argument is always a string, called the control string. The control string in this example is

"\nA marathon is %f kilometers.\n\n"

It is the first argument to the function pri ntfO. Inside this string is the conversion specification, or format, %f. The formats in a control string, if any, are matched with the remaining arguments in the pri ntfO function. In this case, %f is matched '\vith the argument kilometers. Its effect is to print the value of the variable kilometers as a

floating-point number and insert it into the print stream where the format %f occurs . •

Certain words, called keywords are reserved and CalIDOt be used by the programmer as names of variables. For example, i nt, float, and double are keywords. A table of keywords appears in Section 2.4, "Keywords," on page 77. Other names are knowll to the C system and normally would not be redefined by the programmer. The name pri ntf is an example. Because pri ntf is the name of a function in the standard library, it usually is not used as the name of a variable.



1.4 'V The Use of #defi ne and #i ncl ude 13

~ decimal point in a number indicates that it is a floating-point constant rather than an mteger constant. Thus, the numbers 37 and 37.0 would be treated differently in a progran:. Although there are three floating types-float, double, and long doub 1 e and varIables can be declared to be of any of these types, floating constants are auto matically of type dou b 1 e.

Expressions typi~ally are f~und on the right side of assignment operators and as arguments to functlOns. The SImplest expressions are just constants such as 385 d 1760.0, which were used in the previous program. The name of a variable itself anb 'd d . d can e conSI ere an expreSSlOn, an meaningful combinations of operators with variables and constants are also expressions.

!he evalu.a:i~n of expr~ssions can involve conversion rules. This is an important pomt. The dlVIslOn of two mtegers results in an integer value, and any remainder is dis carded. T~us, for exam~l~, the expre~sion 7/2 has i nt value 3. The expression 7.0/2, ~owever, IS a doubl.e dlv~ded by an lnt. When the expression 7.0/2 is evaluated, the value of the expresslOn 2.1S automatically converted to a doubl e, causing 7.0/2 to have the value 3.5. In the prevlOus program, suppose that the statement

kilometers 1.609 \* (miles + yards / 1760.0);

changed to

= 1.609 \* (miles + yards / 1760);

lea~s to a progra~ bug. Because the variable yards is of type i nt and has value the mteger expresslOn

s integer division, and the result is the i nt value O. This is not what is wanted. Use "0 constant 1760.0, which is of type double, corrects the bug.

**The Use of #defi ne and #i ncl ude**

C compi~er ~as a preprocessor built into it. Lines that begin '\vith a # are called pre- ~eSSIYla dIrectIVes. If the lines

LIIVJIT 100

PI 3.14159

14 Chapter 1 V An Overview of C

occur in a file that is being compiled, the preprocessor first changes all occurrences of the identifier LIMIT to 100 and all occurrences of the identifier PI to 3.14159, except in quoted strings and in comments. The identifiers LIMIT and PI are called *symbolic constants.* A #defi ne line can occur anywhere in a program. It affects only the lines in the file that come after it.

Normally, all #defi ne lines are placed at the beginning of the file. By convention, all identifiers that are to be changed by the preprocessor are '\tv-ritten in capital letters. The contents of quoted strings are never changed by the preprocessor. For example, in the statement

pri ntf(" PI == %f\n ", PI);

only the second PI will be changed by the above #defi ne directives to the preproces sor. The use of symbolic constants in a program make it more readable. More impor

1.4 V The Use of #defi ne and #i ncl ude 1 5

standard header file *stdio.h* should be included. This file contains the declarations, or more specifically, the function prototypes, of these functions. (See Section 1. 7, "Func tions," on page 29, for further discussion.)

The Santa Cruz campus of the University of California overlooks the Monterey Bay on the Pacific Ocean and some of the ocean just to the northwest of the bay. We like to call this part of the ocean that is visible from the campus the "Pacific Sea." To illustrate how the #i nc 1 ude facility works, we will 'write a program that prints the area of the Pacific Sea in various units of measure. First, we create a header file and put in the following lines:

file pacificsea.h

#include <stdio.h>

tantly, if a constant has been defined symbolically by means of the #defi ne facility and used throughout a program, it is easy to change it later, if necessary. For example, in physics the letter c is often used to designate the speed of light, which is apprOximately 299792.458 lan/sec. If we write

#define #define #define #define #define

AREA

SO-MILES\_PE~SO-KILOMETER

SO-FEET\_PE~SO-MILE SO-INCH ES\_PER\_SQ\_FOOT ACRES\_PER\_SO-MILE

2337

0.3861021585424458 (5280 ,~ 5280)

144

640

#define C 299792.458 /\* speed of light in km/sec \*/

and then use C throughout thousands of lines of code to represent symbolically the constant 299792.458, it will be easy to change the code when a new phYSical experi ment produces a better value for the speed of light. All the code is updated by simply changing the constant in the #defi ne line.

In a program, a line such as

#include "my\_file.h"

we write the function main 0 in a .c file.

/\* Measuring the Pacific Sea. \*/ #include "pacificsea.h"

int main(void)

{

double pacific\_sea AREA; *t* l' in sq kilometers 1,/

is a preprocessing directive that causes a copy of the file *my\_file.h* to be included at this point in the file when compilation occurs. A #i ncl ude line can occur anywhere in a file,

const int

acres, sq\_miles, sq\_feet, sq\_inches;

though it is typically at the head of the file. The quotes surrounding the name of the file are necessary. An include file, also called a *header file,* can contain #defi ne lines and other #i ncl ude lines. By convention, the names of header files end in *.h.*

The C system provides a number of standard header files. Some examples are *stdio.h, string.h,* and *math.h.* These files contain the declarations of functions in the standard library, macros, structure templates, and other programming elements that are com monly used. As we have already seen, the preprocessing directive

#include <stdio.h>

causes a copy of the standard header file *stdio.h* to be included in the code when compi lation occurs. In ANSI C, whenever the functions pri ntfO or scanfO are used, the 

printf("\nThe Pacific Sea covers an area"); printf(" of %d square kilometers.\n", pacificsea); sq\_miles == SO-MILES\_PER\_SO-KILOMETER \* pacific\_sea; sq\_feet SO-FEET\_PER\_SO-MILE \* sq\_miles; sq\_inches == SO-INCHES\_PER\_SO-FOOT \* sq\_feet; acres = ACRES\_PER\_SO-MILE -{, sq\_mi 1 es;

printf("In other units of measure this is:\n\n"); printf("%22.7e acres\n", acres);

printf("%22.7e square miles\n", sq\_miles); printf("%22.7e square feet\n", sq\_feet); printf("%22.7e square inches\n\n", sq\_inches); return 0;

-----------~--~~~~~~ --~~-------

16 Chapter 1 T An Overview of C

Now our program is written in two files, a .h file and a .c file. The output of this pro gram is

The Pacific Sea covers an area of 2337 square kilometers. In other units of measure this is:

5.7748528e+05 acres

9.0232074e+02 square miles

2.515525ge+10 square feet

3.6223572e+12 square inches

The new programming ideas are described in the following dissection table. • Dissection of the pacific\_sea Program  III #include "pacific\_sea.h"

This #i ncl ude line is a preprocessing directive. It causes a copy of the *filepacificsea.h* to be included when the program is compiled. Because this file contains the line

#include <stdio.h>

the preprocessor expands the line in turn and includes a copy of the standard header file *stdio.h* in the code as well. We have included *stdio.h* because we are using pri ntfO. Five symbolic constants are defined in *paci(icsea.h.*

III #define AREA 2337

This #defi ne line is a preprocessing directive. It causes the preprocessor to replace all occurrences of the identifier AREA by 2337 in the rest of the file. By convention, capital letters are used for identifiers that will be changed by the preprocessor. If at some future time a new map is made and a new figure for the area of the Pacific Sea is com

puted, only this line needs to be changed to update the program.

III #define 0.3861021585424458

The floating constant 0.3861021585424458 is a conversion factor. The use of a sym bolic name for the constant makes the program more readable.

1.4 T The Use of #defi ne and #i ncl ude 17

III #define *(5280 t, 5280)*

The preprocessor changes occurrences of the first sequence of characters into the sec ond. If a reader of this program knows that there are 5280 feet in a mile, then that reader will quickly recognize that this line of code is correct. Instead of (5280 ,', 5280), we could have vVTitten 27878400; because C compilers expand constant expres sions during compilation, run-time efficiency is not lost. Although the parentheses are not necessary, it is considered good programming practice to use them. For technical reasons parentheses are often needed around symbolic expressions. (See Section 8.3, "Macros with Arguments," on page 368.)

l1li canst int pacific\_sea = AREA; /\* in sq kilometers \*/

When compiled, the preprocessor first changes AREA to 2337. The compiler then inter prets this line as a declaration of the identifier paci fi c\_sea. The variable is declared as type i nt and initialized to the value 2337. The keyword canst is a type qualifier that has been newly introduced by ANSI C. It means that the associated variable can be ini tialized, but cannot thereafter have its value changed. (See exercise 18, on page 65.) On some systems this means that the variable may be stored in ROM (read-only memory).

III double acres, sq\_miles, , sq\_inches;

These variables are defined to be of type double. In ANSI C, floating types are float, doub 1 e, and long doub 1 e; long double does not exist in traditional C. Each of these types is used to store real values. Typically. a fl oa t vyill store 6 significant digits and a doub 1 e will store 15 significant digits. Along double will store at least as many signif

icant digits as a daub 1 e. (See Section 3.6, "The Floating Types," on page 119.) III printf("%22.7e acres\n", acres);

This statement causes the line

5.7748528e+05 acres

to be printed. The number is vvritten in scientific notation and is interpreted to mean 5.7748528 x 105. Numbers written this way are said to be written in an e-format. The conversion specification %e causes the system to print a floating expression in an e-for mat vvith default spacing. A format of the form %m.ne, where m and n are positive inte gers, causes the system to print a floating expression in an e-format in *m* spaces total, with *n* digits to the right of the decimal paint. (See Section 11.1, "The Output Function pri ntfO," on page 493.) •

18 Chapter 1." An Overview of C 

**1.5 The Use of printf() and scanf()**

The function pri ntfO is used for output. In an analogous fashion, the function seanfO is used for input. (The fin pri ntf and seanf stands for *formatted.)* Techni cally, these functions are not part of the C language, but rather are part of the C system. They exist in a library and are available for use wherever a C system resides. Although the object code for functions in the library is supplied by the C system, it is the respon sibility of the programmer to declare the functions being used. ANSI C has introduced a new and improved kind of function declaration called a *function prototype.* This is one of the most important changes introduced into the language by ANSI C. The function prototypes of functions in the standard library are available in the standard header

1.5 ." The Use of pri ntfO and seanfO

19

printfO conversion characters

| Conversion  character How the corresponding argument is printed  c as a character  d as a decimal integer  e as a floating-point number in scientific notation f as a floating-paint number  9 in the e-format or f-format, whichever is shorter 5 as a string |
| --- |

files. In particular, the function prototypes for pri ntfO and seanfO are in *stdio.h.* Thus, this header file should be included whenever the function pri ntfO or scanfO is used. (See Section 1.7, "Functions," on page 29.)

Both pri ntfO and seanfO are passed a list of arguments that can be thought of as *controLstrtng* and *other\_arguments*

where *controLstring* is a string and may contain conversion specifications, or formats. A conversion specification begins with a % character and ends vvith a conversion charac ter. For example, in the format %d the letter d is the conversion character. As we have already seen, this format is used to print the value of an integer expression as a decimal integer. To print the letters on the screen, we could use the statement

printf(lfabe lf );

Another way to do this is with the statement

pri ntf("%slf I "abc");

The format %s causes the argument If abc If to be printed in the format of a string. Yet another way to do this is with the statement

printf("%c%c%e lf , 'a', 'b', 'e');

Single quotes are used to designate character constants. Thus, 'a' is the character con stant corresponding to the lowercase letter *a.* The format %e prints the value of an expression as a character. Notice that a constant by itself is considered an expression.

When an argument is printed, the place where it is printed is called its *field* and the number of characters in its field is called its *field width.* The field width can be speCified in a format as an integer occurring between the % and the conversion character. Thus, the statement

pri ntf("%c%3e%5c\n", I A I, 'B', • C');

vvill print

ABC

The function scanfO is analogous to the function pri ntfO but is used for input rather than output. Its first argument is a control string having formats that corre spond to the various ways the characters in the input stream are to be interpreted. The other arguments are *addresses.* ConSider, for example, the statement

scanf("%d", &x);

The format %d is matched with the expression &x, causing seanfO to interpret charac ters in the input stream as a decimal integer and store the result at the address of x. Read the expression &x as "the address of x" because & is the address operator.

When the keyboard is used to input values into a program, a sequence of characters is typed, and it is this sequence of characters, called the input stream, that is received by the program. If 1337 is typed, the person typing it may think of it as a decimal inte ger, but the program receives it as a sequence of characters. The scanfO function can be used to convert a string of decimal digits into an integer value and to store the value at an appropriate place in memory.



20 Chapter 1" An Overview of C

. The functio~l scanfO returns an int value that is the number of successful conver SlO~s accomphshe~ or the system defined end-of-value. The function pri ntfO returns an 1 nt value that IS the number of characters printed or a negative value in case of an error.

scanfO conversion

Conversion

character How characters in the input stream are converted

c character

d decimal integer

f floating-point number (float)

If or LF floating-point number (double)

5 string

~he d.etails ~oncer~~g pri ntfO and scanfO are found in Section 11.1, "The Output ;unctlOn p rl ntf 0, \_ on page 493, and in Section 11.2, "The Input Function scanfO," n page 499. Here, we only want to present enough information to get data into and out

of the mach" .. me m a mImmally acceptable way. The following program reads in three  chharacters and some numbers and then prints them out. Notice that variables of type c ar are used to store character values.

In file echo.c

#include <stdio.h>

int main(void)

{

1.6 " Flow of Control 21

If we compile the program, run it, and type in ABC 3 55 77.7, then this is what appears on the screen:

Input three characters,

an int, a float, and a double: ABC 3 55 77.7

Here is the data that you typed in:

ABC 3 5.S00000e+01 7. 770000e+01

When reading in numbers, scanfO will skip white space (blanks, newlines, and tabs), but when reading in a character, white space is not skipped. Thus, the program will not run correctly with the input AB C 3 55 77. l. The third character read is a blank, which is a perfectly good character; but then scanfO attempts to read C as a decimal integer, which causes difficulties.

**1.6 Flow of Control**

Statements in a program are normally executed in sequence. However, most programs require alteration of the normal sequential flow of control. The if and i f-e 1 se state ments provide alternative actions, and the whi 1 e and for statements provide looping mechanisms. These constructs typically require the evaluation of logical expressions, expressions that the programmer thinks of as being either *true* or *false.* In C, any non zero value is considered to represent *true,* and any zero value is considered to repre sent *false.*

The general form of an if statement is

char

int

float double

c1, c2, c3; i ;

X' ,

y;

if *(expr)*

*statement*

If *expr* is nonzero *(true),* then *statement* is executed; otherwise, it is skipped. It is impor

pri~tf(':\n%s\n%s", "Input three characters, II

an lnt, a float and a double' ").

sC~nf(II:c%c%c%d~f%lf", &c1, &c2, &c3, &ti, &x, &y); pr:ntf(II\nHere 1S the data that you typed in:\n"); pn ntf( %3c%3c%3c%5d%17e%17e\n\n" c1 c2 c3 . return 0; , , , ,1, X, y) ; }

tant to recognize that an if statement, even though it contains a statement part, is itself a single statement. Consider as an example the code

a = 1;

if (b == 3)

a = 5;

pri ntf("%d", a);

The symbols == represent the *is equal to* operator. In the code above, a test is made to see if the value of b is equal to 3. If it is, then a is assigned the value 5 and control

22 Chapter 1 T An Overview of C 

passes to the pri ntfO statement, causing 5 to be printed. If, however, the value of b is not 3, then the statement

a = 5;

is skipped and control passes directly to the pri ntfO statement, causing 1 to be printed. In C, logical expressions have either the i nt value 1 or the i nt value O. Con sider the logical expression

b == 3

This expression has the i nt value 1 *(true)* if b has the value 3; otherwise, it has the i nt value 0 *(f'alse).*

A group of statements surrounded by braces constitutes a *compound statement.* Syn tactically, a compound statement is itSelf a statement; a compound statement can be used anywhere that a statement can be used. The next example uses a compound state ment in place of a simple statement to control more than one action:

if Ca == 3) {

b 5;

e = 7;

1.6 T Flow of Control 23

if (ent 0) {

a == 2 ;

b == 3;

e 5 ;

} else {

a -1;

b -2;

e -3;

}

printf(lf%d", a + b + e);

This causes 10 to be printed if ent has value 0, and causes -6 to be printed otherwise. Looping mechanisms are very important because they allow repetitive actions. The following program illustrates the use of a whi 1 e loop:

In file consecutive\_sums.c

#include <stdio.h>

int mainCvoid)

{

}

Here, if a has value 3, then two aSSignment statements are executed; if a does not have value 3, then the two statements are skipped.

An i f-e 1 se statement is of the form

if *Cexpr)*

i nt i = 1, sum

while Ci <= 5) { sum += i;

++i;

}

0;

*statement]*

else

*statement2*

It is important to recognize that the whole construct, even though it contains state ments, is itself a single statement. If *expr* is nonzero *(true),* then *statement]* is executed; otherwise *statement2* is executed. As an example, consider the code

printfC"sum %d\n", sum);

return 0;

}

• • Dissection of **the** *consecutive\_sum *Program

• while C; <= 5) {

sum += i;

++i;

}

This construct is a whi 1 e statement, or whi 1 e loop. The symbols <= represent the *less than or equal to* operator. A test is made to see if i is less than or equal to 5. If it is, the group of statements enclosed by the braces { and} is executed, and control is passed

24 Chapter 1 T An *Overview* of C

back to the beginning of the whi 1 e loop for the process to start over again. The whi 1 e loop is repeatedly executed until the test fails-that is, until i is not less than or equal

1.6 'f Flow of Control 25

whil e (expr)

statement

to 5. When the test fails, control passes to the statement immediately following the whi 1 e statement, which in this program is a pri ntfO statement. 

III sum += i;

This is a new kind of assignment statement. It causes the stored value of sum to be incremented by the value of i. An equivalent statement is

sum sum + i;

The variable sum is assigned the old value of sum plus the value of i. A construct of the form

variable op= expr

where op is an operator such as +, -, 1<, or / is equivalent to

variable variable op (expr)

II ++i;

C uses ++ and to increment and decrement, respectively, the stored values of vari ables. The statement

++i ; is equivalent to i = i + 1;

In a similar fashion, --i will cause the stored value of i to be decremented. (See Section 2.10, "Increment and Decrement Operators," on page 85, for further discussion of these operators.) •

A hand simulation of the program shows that the whi 1 e loop is executed five times with i taking on the values 1, 2, 3, 4, 5 successively. When control passes beyond th~ whi 1 e statement, the value of i is 6, and the value of sum is

1+2+3+4+5 which is equal to 15

This is the value printed by the pri ntfO statement.

The general form of a whi 1 e statement is

where statement is either a simple statement or a compound statement. When the whi 1 e statement is executed, expr is evaluated. If it is nonzero (true), then statement is executed and control passes back to the beginning of the whi 1 e loop. This process con tinues until expr has value 0 (false). At this paint, control passes on to the next state ment. In C, a logical expression such as i <= 5 has i nt value 1 (true) if i is less than or equal to 5, and has i nt value 0 (false) otherwise. 

Another looping construct is the for statement. (See Section 4.9, "The for State ment," on page 167, for a more complete discussion.) It has the form

for (exprl; expr2; expr3)

statement

If all three expressions are present, then this is equivalent to

exprl;

whi 1 e (expr2) {

statement

expr3;

}

Typically, exprl performs an initial assignment, expr2 performs a test, and expr3 incre ments a stored value. Note that expr3 is the last thing done in the body of the loop. The for loop is repeatedly executed as long as expr2 is nonzero (true). For example,

for (i = 1; i <= 5; ++i)

sum += i;

This for loop is equivalent to the whi 1 e loop used in the last program. Our next program illustrates the use of an i f-e 1 se statement within a fo r loop . Numbers are read in one after another. On each line of the output we print the count and the number, along with the minimum, maximum, sum, and average of all the num bers seen up to that point. (See exercise 16, on page 65, through exercise 18, on page 65, for further discussion concerning the computation of the average.)

,

26 Chapter 1 V An Overview of C 

the t'ollovving appears on the screen:

1.6 V Flow of Control

27

/\* Compute the minimum, maximum, sum, and average.

#include <stdio.h>

#include <stdlib.h>

i nt mai n(voi d)

Count Item Min Max

1 3.0 3.0 3.0 2 -5.0 5.0 3.0 3 7.0 5.0 7.0

Sum

3.000 -2.000 5.000

Average

3.000

-1.000 1.667

{

int

double

i . ,

x, min, max, sum, avg;

The use of the symbol < in the command

if (scanf("%lf", &x) 1= 1) {

printf("No data found bye!\n");

exit(1);

}

min = max = sum = avg = x;

printf("%5s%9s%9s%9s%12s%12s\n%5s%9s%9s%9s%12s%12s\n\n", "Count", "Item", "Min", "Max", "Sum", "Average", " n, II lI, 11 II, If If, II B, **n ") ;** printf("%5d%9.lf%9.lf%9.1f%12.3f%12.3f\n", 1, x, min, max, sum, avg);

running\_sum < data 

causes the input to be redirected. The program *running\_sum* takes its input from the standard input file, which is normally connected to the keyboard. The operating sys tem, however, has redirected the input to the file *data.* In this context, the symbol < is thought of as a left pointing arrow. (See Section 1.10, "Operating System Consider ations," on page 53, for further discussion.)

• Dissection of the running\_sum Program 

for (i = 2; scanf("%lf", &x) if (x < mi n)

min = x;

else if (x > max)

max = x;

sum += x;

avg = sum / i;

1; ++i) {

III if (scanf("%lf", &x) 1 1) { printf("No data found byel\ntl); exit(1);

}

pri ntf("%5d%9.lf%9.lf%9.1f%12.3f%12. 3f\n" , i, x, min, max, sum, avg); } return 0;

}

Recall that scanf 0 returns as an i nt the number of successful conversions per formed. If scanfO is unable to make a conversion, then we print a message and exit the program. The function exi to is in the standard library, and its function prototype is in *stdlib.h.* When ex itO is invoked, certain housekeeping tasks are performed and

This program has been designed to read numbers from a file. We can type them in from the keyboard, but if we do this, then what appears on the screen will not be formatted correctly. To test this program, we compile it and put the executable code in *running\_sum.* Then we create a file called *data* and put the following numbers in it:

3 -5 7 -9 11 -13 15 -17 19 -21 

Now, when we give the command

running\_sum < data

the program is terminated. This function takes a single argument of type i nt that, by convention, is zero if the programmer considers the exit to be normal, and is nonzero otherwise.

III printf("%5s%9s%9s%9s%12s%12s\n%5s%9s%9s%9s%12s%12s\n\n", "Count", "Item", "Min", "Max", "Sum", "Average",

II 11 If II II II tI f1 it H 11

This statement prints headings. The field widths in the formats have been chosen to put headings over columns.

28 Chapter 1 'f An Overview of C 

III printfCI%5d%9.lf%9.lf%9.lf%12.3f%12.3f\n",

1, x, min, max, sum, avg);

After the headings, this is the first line to be printed. Notice that the field widths here match the field widths in the previous pri ntfO statement.

III for Ci = 2; scanfCI%lf", &x) == 1; ++i) {

The variable i is initially assigned the value 2. Then a test is made to see if the logical expression

scanfCI%lf", &x) == 1

is true. If scanfO can read characters from the standard input stream, interpret them as a doub 1 e (read 1 f as "long float"), and place the value at the address of x, then a suc cessful conversion has been made. This causes scanfO to return the i nt value 1, which in turn makes the logical expression true. As long as scanfO can continue to read characters and convert them, the body of the fo r loop will be executed repeatedly. The variable i is incremented at the end of the body of the loop.

III if Cx < min)

min = x;

else if Cx > max)

max = x;

This construct is a single i f-e 1 se statement. Notice that the statement part following the else is itself an if statement. Each time through the loop this i f-e 1 se statement causes the values for mi n and max to be updated, if necessary. •

1.7'f Functions 29

**1.7 Functions**

The heart and soul of C programming is the function. A function represents a piece of code that is a building block in the problem-solving process. All functions are on the same external level; they cannot be nested one inside another. A C program consists of one or more functions in one or more files. (See Section 5.8, "Developing a Large Pro

gram," on page 209.) Precisely one of the functions is a mai nO function, where execu tion of the program begins. Other functions are called from within mai nO and from within each other.

Functions should be declared before they are used. Suppose, for example, that we want to use the function powO, called the power function, one of many functions in the mathematics library available for use by the programmer. A function call such as powCx, y) returns the value of x raised to the y power. To give an explicit example, powC2. 0, 3.0) yields the value 8.0. The declaration of the function is given by

double powCdouble x, double y);

Function declarations of this type are called *function prototypes.* An equivalent function prototype is given by

double powCdouble, double);

Identifiers such as x and y that occur in parameter type lists in function prototypes are not used by the compiler. Their purpose is to provide documentation to the program mer and other readers of the code .

A function prototype tells the compiler the number and type of arguments to be passed to the function and the type of the value that is to be returned by the function. ANSI C has added the concept of function prototype to the C language. This is an important change. In traditional C, the function declaration of powO is given by

double powO; /\* traditional style \*/

Parameter type lists are not allowed. ANSI C compilers will still accept this style, but function prototypes, because they greatly reduce the chance for errors, are much pre ferred. (See exercise 5, on page 236, in Chapter 5, "Functions.")

A function prototype has the following general form:

*type function\_nameCparameter type list) ;*

**--**

30 Chapter 1" An Overview of C 

The *parameter type list* is typically a list of types separated by commas. Identifiers are optional; they do not affect the prototype. The keyword voi d is used if a function takes no arguments. Also, the keyword voi d is used if no value is returned by the function. If a function takes a variable number of arguments, then ellipses ... are used. For exam

ple, the function prototype

i nt pri ntfCconst char ~'format, ... );

can be found in *stdio.h.* (See exercise 14, on page 63.) This information allows the com



}

1.7 " Functions

31

for Ci = 2; i <= n; ++i) {

scanfC"%f", &x);

max = maximumCmax, x);

min = minimumCmin, x);

} printfC"\n%s%l1.3f\n%s%11.3f\n\n", "Maximum value:", max,

"Minimum value:", min);

return 0;

piler to enforce type compatibility. Arguments are converted to these types as if they were follOwing rules of assignment.

To illustrate the use of functions, we set for ourselves the following task:

Creating maxmin Program

1 Print information about the program (this list).

2 Read an integer value for *n.*

float maximumCfloat x, float y) {

if Cx > y)

return x;

else return y;

}

float minimumCfloat x, float y)

3 Read in *n* real numbers. 

{

4 Find minimum and maximum values.

Let us write a program called *maxmin* that accomplishes the task. It consists of three }

functions written in the file *maxmin.c.*

if Cx < y) return x;

else return y;

In file maxmin.c

#include <stdio.h>

void prn\_infoCvoid)

{ printfC"\n%s\n%s\n\n",

"This program reads an integer value for n, and then",

float float

maximumCfloat x, float y);

}

minimumCfloat x, float y);

"processes n real numbers to find max and min values.");

void

prn\_infoCvoid);

To test the program, we give the command

int mainCvoid)

*maxmin*

{

float i, n;

i nt

max, min, x;

Suppose, when prompted, we type in 5 followed by the line

prn\_infoO;

pri ntfC"Input n: ");

scanfC"%d", &n);

printfC"\nlnput %d real numbers: scanfC"%f" , &x);

max = min = x;

737.7799 -11.2e+3 -777 0.001 3.14159

" n) ;

i L

32 Chapter 1.. An Overview of C

Here is what appears on the screen:

This program reads an integer value for n, and then processes n real numbers to find max and min values.

Input n: 5

Input 5 real numbers: 737.7799 -11.2e+3 -777 0.001 3.14159

1.7 .. Functions 33

• int main(void)

{ i nt

float i, n;

max, min, x;

prn\_infoO;

pri ntfC'Input n: ");

scanf("%d", &n);

Maximum value: 737.780

Minimum value: -11200.000

• Dissection of the maxmin Program

• #include <stdio.h>

Variables are declared at the beginning of ma in O. The first executable statement in mainO is



This statement invokes the function prn\_infoO. The function contains a single pri ntfO statement that prints information about the program on the screen. The user responds to the prompt by typing characters on the keyboard. We use scanfO to inter

float float void

maximum(float x, float y); minimum(float x, float y); prn\_info(void);

pret these characters as a decimal integer and to place the value of this integer at the address of n.

• printf("\nInput %d real numbers: ", n);

i nt mai n (voi d)

{

The function prototypes for the functions maxi mumO, mi ni mumO, and prn\_ i nfoO occur at the top of the file after any #i ncl ude lines and #defi ne lines. The first two function prototypes tell the compiler that the functions maxi mum 0 and mi ni mum 0 ea::h take t,wo arguments of type float and each return a value of type float. The thlrd functIOn prototype tells the compiler that prn\_ i nfoO takes no arguments and returns no value. Note that for the first two function prototypes we could just as well have written

float

float maximum(float, float)· . . , mlnlmum(float, float);

The co:upiler does not make any use of parameters such as x and y in function proto types. rhe parameters serve only as documentation for the reader of the code.

scanf("%f", &x); 

max = min = x;

The user is asked to input n real numbers. The first real number is read and its value is placed at the address of x. Because the assignment operator associates from right to left

max min = x; is equivalent to max = (mi n = x);

Thus, the value x is assigned first to mi n and then to max. (See Section 2.9, "Precedence and Associativity of Operators," on page 83.)

• for (i = 2; i <= n; ++i) {

scanf("%f", &x);

max max;mum(max, x);

min = minimum(min, x);

}

Each time through the loop a new value for x is read in. Then the current values of max and x are passed as arguments to the function maxi mum 0, and the larger of the two values is returned and assigned to max. Similarly, the current values of mi n and x are passed as arguments to the function mi ni mumO, and the smaller of the two values is returned and assigned to mi n. In C, arguments to functions are always passed *by* value.



34 Chapter 1 'Y An Overview of C

This means that a copy of the value of each argument is made, and it is these copies that are processed by the function. The effect is that variables passed as arguments to functions are *not changed* in the calling environment.

.. float maximum(float x, float y) {

1.7'Y Functions 35

A retu rn statement causes control to be passed back to the calling environment. If an expression follows the keyword return, then the value of the expression is passed back as well.

.. float minimumefloat x, float y)

}

if (x > y) return x;

else return y;

{ }

if ex < y) return x;

else return y;

This is the function definition for the function maxi mum O. It specifies explicitly how the function will act when it is called, or invoked. A function definition consists of a header and a body. The header is the code that occurs before the first left brace {. The body consists of the declarations and statements between the braces { and}. For this function definition the header is the line

float maximum(float x, float y)

The first keyword float in the header tells the compiler that this function is to return a value of type float. The parameter list consists of the comma-separated list of identi fier declarations within the parentheses e and) that occur in the header to the function definition. Here, the parameter list is given by

float x, float y

The identifiers x and yare formal parameters. Although we have used the identifiers x and y both here and in the function ma in 0, there is no need to do so. There is no rela tionship, other than a mnemonic one, between the x and y used in maxi mum 0 and the x and y used in ma in O. Parameters in a function definition can be thought of as place holders. When expressions are passed as arguments to a function, the values of the expressions are associated with these parameters. The values are then manipulated according to the code in the body of the function definition. Here, the body of the func tion definition consists of a single i f-el se statement. The effect of this statement is to return the larger of the two values x and y that are passed in as arguments.

.. return x;

This is a retu rn statement. The general form of a retu rn statement is return; or retu rn *expr;*

The function definition for mi n i mum 0 comes next. Note that the header to the function definition matches the function prototype that occurs at the top of the file. This is a common programming style.

.. void prn\_infoevoid)

{

This is the function definition for prn\_ i nfoO. The first voi d tells the compiler that • •

this function returns no value, the second that this function takes no arguments.

**Call-by-Value**

In C, arguments to functions are always passed by value. This means that when an expression is passed as an argument to a function, the expression is evaluated, and it is this value that is passed to the function. The variables passed as arguments to func tions are *not changed* in the calling environment. Here is a program that illustrates this:



36 Chapter 1... An Overview of C

In file no\_change.c

#include <stdio.h>

int main(void)

{ int a = 1;

void try\_to\_change\_it(;nt);

1.8 ... Arrays, Strings, and Pointers 37

**Arrays**

Arrays are used when many variables, all of the same type, are desired. For example, the declaration

int a[3];

allocates space for the three-element array a. The elements of the array are of type i nt

pri ntf("%d\n" , a); try\_to\_change\_;t(a);  pri ntf("%d\n" , return 0;

a);

}

/\* 1 ;5 /\* 1 is

printed \*/

printed again! \*/

and are accessed as a [0], a [1], and a [2]. The index, or subscript, of an array always starts at O. The next program illustrates the use of an array. The program reads in five scores, sorts them, and prints them out in order.

In file scores.c

void 

#include <stdio.h>

{ }

a = 777;

#define CLASS\_SIZE 5 int main(void)

{

When a is passed as an argument, the expression a is evaluated to produce a value that we can think of as a copy of a. It is this copy, rather than a itself, that is passed to the function. Hence, in the calling environment the variable a does not get changed.

Tbis argument-passing convention is known as *call-by-value.* To change the value of a variable in the calling environment, other languages provide *call-by-reference.* In C, to get the effect of call-by-reference, pointers must be used. (See Section 6.3, "Call-by-Ref erence," on page 252.)

int i, j, score[CLASS\_SIZEJ. sum = 0, tmp;

pri ntf("Input %d scores: ", CLASS\_SIZE); for (i = 0; i < CLASS\_SIZE; ++i) { scanf("%d", &score[i]);

sum += scorer;];

}

**1.8 Arrays, Strings, and Pointers**

In C, a string is an array of characters, and an array name by itself is a pointer. Because

for (i 0; i < CLASS\_SIZE - 1; ++i) for (j = CLASS\_SIZE - 1; j > i; if (score[j-1J < score[jJ) { tmp = score[j-l];

score[j-1] = score[jJ;

score[j] = tmp;

}

pr;ntf("\nOrdered scores:\n\n"); for (; = 0; i < CLASS\_SIZE; ++i)

/\* bubble sort \*/ ) /\* check the order \*/

of tbis, the concepts of arrays, strings, and pointers are intimately related. A pointer is just an address of an object in memory. C, unlike most languages, provides for pointer arithmetic. Because pointer expressions of great utility are possible, pointer arithmetic is one of the strong points of the language.

}

pr;ntf(" score[%d] =%5d\n", i, score[i]); printf("\n%18d%s\n%18.lf%s\n\n",

sum, II ;5 the sum of all the scores",

(double) sum / CLASS\_SIZE, " ;5 the class average"); return 0;

If we execute the program and enter the scores 63, 88, 97, 53, 77 when prompted, we will see on the screen

38 Chapter 1 V An Overview of C

1.8 V Arrays, Strings, and Pointers 39

Input 5 scores: Ordered scores: score [0J

score [lJ

score [2J

score [3J

63

97

88

77

63

88 97 53 77

In C, ~ string is an arr~y of characters. In this section, in addition to illustrating the use of strmgs, we want to mtroduce the use of getcharO and putcharO. These are mac ros defined in stdio.h. Although there are technical differences, a macro is used in the

score[4]

53

378

75.6

is the sum of all the scores is the class average

same way a function is used. (See Section 8.7, "The Macros in stdio.h and ctype.h," on page 382.) The macros getcharO and putcharO are used to read characters from the keyboard and to print characters on the screen, respectively.

A bubble sort is used in the program to sort the scores. This construction is typically done with nested for loops, with a test being made in the body of the inner loop to check on the order of a pair of elements. If the elements being compared are out of order, their values are interchanged. Here, this interchange is accomplished by the code

tmp = score[j-lJ;

score[j-lJ = score[jJ;

score[j] =: tmp;

In the first statement, the variable tmp is used to temporarily store the value of score [j-l]. In the next statement, the value of score [j-1J stored in memory is being ovemTitten with the value of score [jJ. In the last statement, the value of score [j] is being overwritten with the original value of score [i], which is now in tmp. Hand simu 

lation of the program with the given data will show the reader why this bubble sort con struct of two nested fa r loops achieves an array with sorted elements. The name bubble sort comes from the fact that at each step of the outer loop the desired value among those left to be worked over is bubbled into position. Although bubble sorts are easy to code, they are relatively inefficient. Other sorting techniques execute much faster. This is of no concern when sorting a small number of items infrequently, but if the number of items is large or the code is used repeatedly, then efficiency is, indeed,

Our next ~rogram ~tores a line typed in by the user in an array of characters (a string) and then prmts the lme backwards on the screen. The program illustrates how charac ters in C can be treated as small integers.

/\* Have a nice day! \*/

#include <ctype.h>

#include <stdio.h>

#define MAXSTRING 100

int main(void)

{

char c, name[MAXSTRING];

int i, sum = 0;

printf("\nHi! What is your name? ");

for (i = 0; (c = getchar()) != '\n'; ++i) {

name[iJ c;

if Ci sal pha(c))

an important consideration. The expression

}

sum += Cj

(double) sum / CLASS\_SIZE 

which occurs as an argument in the final pri ntfO statement, uses a cast operator. The

effect of (double) sum is to cast, or convert, the i nt value of sum to a doub 1 e. Because

the precedence of a cast operator is higher than that of the division operator, the cast is

done before division occurs. (See Section 2.9, "Precedence and Associativity of Opera

tors," on page 83.) When a daub 1 e is divided by an i nt, we have what is called a mixed

expression. Automatic conversion now takes place. The i nt is promoted to a doub 1 e,

and the result to the operation is a doubl e. If a cast had not been used, then integer

}

division would have occurred and any fractional part would have been discarded. More

over, the result would have been an i nt, which would have caused the format in the

pri ntfO statement to be in error.



name[i] = '\0';

printf("\n%s%s%s\n%s",

"Nice to meet you ", name ","

If ' , Your name spelled backwards is ");

for (--i; i >= 0; --i)

putchar(name[i])j

printf("\n%s%d%s\n\n%s\n",

"and the letters in your name sum to ", sum, "Have a nice day!");

return 0;

" . tf ,

40 Chapter 1 'f An Overview of C 

If we run the program and enter the name Ali ce B. Carole when prompted, the fol lowing appears on the screen:

Hi! What is your name? Alice B. Carole

Nice to meet you Alice B. Carole.

Your name spelled backwards is eloraC .B ecilA

and the letters in your name sum to 1142.

Have a nice day!

• • Dissection of the *nice\_day* Program  • #include <ctype.h> #include <stdio.h>

The standard header file *stdio.h* contains the function prototype for pri ntfO. It also contains the macro definitions for getcharO and putcharO, which will be used to read characters from the keyboard and to write characters to the screen, respectively. The standard header file *ctype.h* contains the macro definition for i sal phaO, which will be used to determine if a character is *alphabetic-that* is, if it is a lower- or upper

case letter. 

• #define MAXSTRING 100

The symbolic constant MAXSTRING will be used to set the size of the character array name. We are making the assumption that the user of this program will not type in more than 99 characters. Why 99 characters? Because the system will add the' \0' as one extra guard character terminating the string.

• char i nt c, name[MAXSTRINGJ;

i, sum 0;

The variable c is of type char. The identifier name is of type array of char, and its size is MAXSTRING. In C, all array subscripts start at O. Thus, name [0J, name [1], ... , name [MAXSTRING - 1J are the elements of the array. The variables i and sum are of type i nt; sum is initialized to O.

• printf("\nHi! What is your name? "); 

This is a prompt to the user. The program now expects a name to be typed in followed by a carriage return.

1.8 'f Arrays, Strings, and Pointers 41

• (c = getchar()) != '\n'

ThiS expression consists of two parts. On the left we have

(c getchar 0)

Unlike other languages, assignment in C is an operator. (See Section 2.11, "Assignment Operators," on page 87.) Here, getcharO is being used to read a character from the keyboard and to assign it to c. The value of the expression as a whole is the value of whatever is assigned to c. Parentheses are necessary because the order of precedence of the operator is less than that of the ! = operator. Thus,

c = getchar() != '\n' is equivalent to c = (getcharO != '\n')

which is syntactically correct, but not what we want. In Section 2.9, "Precedence and Associativity of Operators," on page 83, we discuss in detail the precedence and asso ciativity of operators.

• for (i = 0; (c = getchar()) != '\n'; ++i) {

name[iJ = c;

if (i sal pha(c))

sum += c;

}

The variable i is initially assigned the value O. Then getcha r 0 gets a character from the keyboard, assigns it to c, and tests to see if it is a newline character. If it is not, the body of the for loop is executed. First, the value of c is assigned to the array element name[iJ. Next, the macro isalphaO is used to determine whether c is a lower- or uppercase letter. If it is, sum is incremented by the value of c. As we will see in Section 3.3, "Characters and the Data Type char," on page 111, a character in C has the integer value corresponding to its ASCII encoding. For example, 'a' has value 97, 'b I has value 98, and so forth. Finally, the variable i is incremented at the end of the for loop. The for loop is executed repeatedly until a newline character is received.

• name[i] '\0' ;

After the fa r loop is finished, the null character \0 is assigned to the element name [i ]. By convention, all strings end with a null character. Functions that process strings, such as pri ntfO, use the null character \0 as an end-of-string sentineL We now can think of the array name in memory as

42 Chapter 1 T An Overview of C

o 1 2 3 4 5 6 7 8 **9 10 11 12 13 14 15 16 99** 

III printf("\n%s%s%s\n%s" , "Nice to meet you ", name, ".",

"Your name spelled backwards is H);

43

1.8 T Arrays, Strings, and Pointers

#include <stdio.h>

#include <string.h>

#define MAXSTRING 100

int main(void)

{

Notice that the format %s is used to print the character array name. The elements of the array are printed one after another until the end-of-string sentinel \0 is encountered.

III for (--i; i >= 0; --i)

putchar(name[i]);

If we assume that Ali ce B. Carole followed by a carriage return was typed in, then i has value 15 at the beginning of this for loop. (Do not forget to count from 0, not 1.) After i has been decremented, the subscript corresponds to the last character of the name that was typed in. Thus, the effect of this fo r loop is to print the name on the screen backwards.

III printf("\n%s%d%s\n\n%s\n",

"and the letters in your name sum to ", sum, ".",

}

"Have a ni ce day! ") ;

char c = 'a', 1,p, s [MAXSTRINGJ ;

p = &c;

pri ntf("%c%c%c ", 1,p, '~p + 1, >"p + 2); strcpy(s, "ABC");

pri ntf("%s %c%c%s\n", s, \*s + 6, *t,s* + 7, s + 1); strcpy(s, "she sells sea shells by the seashore"); p s + 14;

for ( ; \*p != '\0'; ++p) {

if ("'p == 'e')

>"p = 'E';

if ("'p == ' ')

i'p = '\n ';

} printf("%s\n", s);

return 0;

We print the sum of the letters in the name typed in by the user, and then we print a final message. After the sum of the letters in the name is printed, a period is printed. Two newlines are used to create a blank line before the final message is printed. Notice that this pri ntfO style allows us to easily visualize what is to appear on the screen. • 

**Pointers**

A pointer is an address of an object in memory. Because an array name is itself a pointer, the uses of arrays and pointers are intimately related. The following program is designed to illustrate some of these relationships:



The output of this program is

abc ABC GHBC

she sells sea shElls

by

thE

sEashorE

• Dissection of the abc Program

III #include <string.h>

The standard library contains many string-handling functions. (See Section 6.11, "String-Handling Functions in the Standard library," on page 272.) The standard header file *string.h* contains the function prototypes for these functions. In this program we will use strcpy() to copy a string.



44 Chapter 1 V An Overview of C

II char c = 'a', \*p, s[MAXSTRING];

The variable c is of type char. It is initialized with the value 'a'. The variable p is of type pointer to char. The string 5 has size MAXSTRING.

II P = &c;

The symbol & is the address operator. The value of the expression &c is the address in memory of the variable c. The address of c is assigned to p. We now think of p as *point ing to* c.

II pr;ntf("%c%c%c ", ~tp, >'<p + 1, 1<p + 2);

The format %c is used to print the value of an expression as a character. The symbol ~< is the dereferencing, or indirection, operator. The expression ~'p has the value of what ever p is pointing to. Because p is pOinting to c and c has the value 'a', this is the value of the expression ~'p and an a is printed. The value of the expression >"p + 1 is one more than the value of >"p, and this causes a b to be printed. The value of the expression "'p + 2 is two more than the value of \*p, and this causes a c to be printed. 

II "ABC"

A string constant is stored in memory as an array of characters, the last of which is the null character \0. Thus, the size of the string constant "ABC" is 4, not 3. Even the null string If II contains one character, namely \0. It is important to realize that string con stants are of type array of char. An array name by itself is treated as a pointer, and this is true of string constants as well.

III strcpy(s, "ABC");

The function strcpy() takes two arguments, both of type pointer to char, which we can think of as strings. The string pointed to by its second argument is copied into memory beginning at the location pointed to by its first argument. All characters up to and including a null character are copied. The effect is to copy one string into another. It is the responsibility of the programmer to ensure that the first argument points to enough space to hold all the characters being copied. After this statement has been exe

cuted, we can think of s in memory as

3 4 **99**

1.8 V Arrays, Strings, and Pointers 45

II pri ntf("%s %c%c%s\n", 5, ''''5 + 6, ~'s + 7, 5 + 1);

The array name 5 by itself is a pointer. We can think of 5 as pointing to 5 [0], or we can think of 5 as being the base address of the array, which is the address of 5 [0]. Printing 5 in the format of a string causes ABC to be printed. The expression "'5 has the value of what 5 is pointing to, which is 5 [0]. This is the character A. Because six letters more than A is G and seven letters more than A is H, the expressions "'5 + 6 and "'5 + 7, printed in the format of a character, cause G and H to be printed, respective!y. ~he expression s + 1 is an example of pointer arithmetic. The value of the exp~esslO.n IS a pointer that points to 5 [lJ, the next character in the array. Thus,s + 1 pl'lnted III the format of a string causes BC to be printed.

II strcpy(s, "she sells sea shells by the seashore"); This copies a new string into s. Whatever was in 5 before gets overwritten. II P = s + 14;

The pointer value s + 14 is assigned to p. An equivalent statement is p &s[14J;

If you count carefully, you will see that p now points to the first letter in the word "she 11 s." Note carefully that even though 5 is a pointer, it is not a pointer variable, but rather a pointer constant. A statement such as

p = s;

is legal because p is a pointer variable, but the statement

s = p;

would result in a syntax error. Although the value of what s points to may be changed, the value of 5 itself may *not* be changed.

II for ( ; \*p != '\0'; ++p) {

if (\*p == 'e')

\*p = 'E';

if ("'p == ' ')

'~p = '\n';

}

As long as the value of what p is pointing to is not equal to the null character, the body of the fa r loop is executed. If the value of what p is pointing to is equal to 'e' I then

46 Chapter 1 T An Overview of C

that value in memory is changed to I E I • If the value of what p is pointing to is equal to I I, then that value in memory is changed to I \n '. The variable p is incremented at the end of the fo r loop. This causes p to point to the next character in the string.

III printf("%s\n", s);

The variable s is printed in the format of a string followed by a newline character. Because the previous for loop changed the values of some of the elements of s, the fol Imving is printed:

III she sells sea shElls

by

thE

sEashorE

In C, arrays, strings, and painters are closely related. To illustrate consider the decla ration

char \*p, s[100J;

This creates the identifier p as a painter to char and the identifier s as an array of 100 elements of type char. Because an array name by itself is a pointer, both p and s are pointers to char. However, p is a variable pointer, whereas s is a constant pointer that points to s [0J. Note that the expression ++p can be used to increment p, but because s is a constant pointer, the expression ++s is wrong. The value of s cannot be changed. Of fundamental importance is the fact that the two expressions

s[iJ and '/«(s + i)

are equivalent. The expression s [i ] has the value of the ith element of the array (count ing from 0), whereas ~'(s + i) is the dereferencing of the expression s + i, a pointer expression that points i character positions past s. In a similar fashion, the two expres sions

p [i ] and ~«(p + i)

are equivalent. • •

1.9 'f Files 47

**=**

**1.9 Files**

Files are easy to use in C. To open the file named *myJile,* the following code can be used:

In file read\_it.c

#include <stdio.h>

int rnain(void)

{ int

FILE c;

''<i fp;

ifp = fopen("rny\_file", "r");

The second line in the body of rna in () declares i fp (which stands for *infile pointer)* to be a pointer to FILE. The function fopen () is in the standard library, and its function prototype is in *stdio.h.* The type FILE is defined in *stdio.h* as a particular structure. To use the construct, a user need not know any details about it. However, the header file must be made available by means of a #i ncl ude directive before any reference to FILE is made. The function fopen 0 takes two strings as arguments, and it returns a pointer to FILE. The first argument is the name of the file, and the second argument is the mode in which the file is to be opened.

Three modes for a file

"r" for read

"w" for "'Tite

'''an for append

When a file is opened for writing and it does not exist, it is created. If it already exists, its contents are destroyed and the writing starts at the beginning of the file. If for some reason a file cannot be accessed, the pointer value NULL is returned by fopenO. After a file has been opened, all references to it are via its file pointer. Upon the completion of a program, the C system automatically closes all open files. All C sys 

tems put a limit on the number of files that can be open simultaneously. Typically, this

48 Chapter 1 "f An Overview of C 

limit is either 20 or 64. When using many files, the programmer should explicitly close any files not currently in use. The library function fc lose 0 is used to close files. Let us now examine the use of files. With text, it is easy to make a frequency analysis of the occurrence of the characters and words making up the text. Such analyses have proven useful in many disciplines, from the study of hieroglyphics to the study of Shakespeare. To keep things simple, we ""ill write a program that counts the occur rences of just uppercase letters. Among our files is one named *chapter],* which is the current version of this chapter. We will write a program called *em\_letters* that will open files for reading and writing to do the analysis on this chapter. We give the command

*ent\_'etters chapter 1 data 1*

to do this. However, before we present our program, let us describe how command line arguments can be accessed from within a program. The C language provides a connec tion to the arguments on the command line. Typically, to use the connection one would code

#include <stdio.h>

int rnainCint argc, char \*argv[])

{

Up until now we have always invoked rnai n 0 as a function with no arguments. In fact,

1.9 "f Files 49

then the program should read file *ehapter2* and 'write to file *data2.* In our program the three words on the command line will be accessible through the three pointers argv [0J, argv [1] I and argv [2].

Here is our program:

In file cnLletters.c

/\* Count uppercase letters in a file. \*/

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{ int c, i, letter[26];

FILE \*ifp, \*ofp;

if Cargc! 3) {

printf("\n%s%s%s\n\n%s\n%s\n\n",

"Usage: ", argv[0] , " infile outfile",

"The uppercase letters in infile will be counted.",

"The results will be written in outfile.");

exit(l) ;

} ifp fopenCargv[l] , "r");

it is a function that can have arguments. The parameter argc stands for *argument count.* Its value is the number of arguments in the command line that was used to exe cute the program. The parameter argv stands for *argument vector.* It is an array of pOinters to char. Such an array can be thought of as an array of strings. The successive elements of the array point to successive words in the command line that was used to execute the program. Thus, argv [0] is a pointer to the name of the command itself. As an example of how this facility is used, suppose that we have written our program and have put the executable code in the file *cnUetters.* The intent of the command line

ofp = fopen(argv[2] , "w"); for Ci 0; i < 26; ++i) letter[i] = 0;

while CCc getcCifp))!= EOF) if Cc >= 'A' && c <= *'Z')* ++letter[c - 'A'];

for Ci 0; i < 26; ++i) { if (i % 6 0)

putc('\n', ofp);

/\* initialize array to zero \*/

/\* find uppercase letters \*/ /\* print results \*/

'A' + i, letter[i]);

fprintf(ofp, "%c:%5d

"

}

*ent\_'etters chapterl data 1*

}

is to invoke the program *ent\_Ietters* with the two file names *chapter]* and *datal* as com

putcC'\n', ofp); return 0;

mand line arguments. The program should read file *chapter]* and write to file *datal.* If we give a different command, say

*ent\_'etters ehapter2 data2*

After we have given the command *enUetters chapterl data 1* this is what we find in the file *datal:*

50 Chapter 1 ... An Overview of C

A: 223 B: 62 C: 193 D: 31 E: 120 F: 89 G: 21 H: 48 I: 304 J: 1 K: 7 L: 50 M: 67 N: 77 0: 91 P: 71 Q: 19 R: 57 s: 196 T: 439 U: 33 V: 4 W: 68 X: 29 Y: 7 Z: 18

Observe that the frequency of the letters in *chapter1* is not what one expects in ordi

1.9 'f' Files Sl

ifp = fopen(argv[l] , "r");

ofp = fopen(argv[2] , "w");

.... Ifwe assume that we have typed the command line

*cnt\_letters chapterl data 1*

nary text.

• Dissection of the *cnCletters* Program 

II int c, i, letter[26];

FILE \*ifp, \*ofp;

The array 1 etter will be used to count the occurrences of the uppercase letters. The variables i fp and ofp are of type *pointer to* FILE. We often use the identifiers i fp and ofp, which stand for *infiZe pointer* and *outfile pointer,* respectively.

II if (argc != 3) {

printf("\n%s%s%s\n\n%s\n%s\n\n",

"Usage: ", argv[0] , " infile outfile",

"The uppe rcase 1 ette rs in i nfi 1 e will be counted.",

"The results will be written in outfile."); 

exit(I);

}

If the number of words on the command line is not three, then the program is being used incorrectly. This causes a message to be printed and the program to be exited. Suppose the following command line is typed:

*ent\_letters chapter 1 abc abc*

Because the line contains four words, argc will have value 4, and this will cause the fol lowing message to appear on the screen:

Usage: cnt\_letters infile outfile

The uppercase letters in infile will be counted.

The results will be written in outfile.



to execute this program, then argv[0] points to the string "cnt\_l etters", argv[1] points to the string "chapter1", and argv[2] points to the string IIdatal". The C sys otem does this automatically. Thus, the file *chapter1* is opened for reading with file pointer i fp referring to it, and the file *datal* is opened for writing with file pointer ofp referring to it. 

letter[i] = 0; /\* initialize array to zero \*/ for (i = 0; i < 26; ++i)

In ANSI C, automatically allocated local array elements need not be initialized to zero. To be sure, the programmer must do it.

II (c = getc(ifp)) != EOF

The function getcO is a macro defined in *stdio.h.* It is similar to getcharO except that it takes as an argument a pointer to FILE. When getc(ifp) is invoked, it gets the next character from the file pointed to by i fp. The identifier EOF stands for *end-of-file.* It is a symbolic constant defined in *stdio.11,* typically by the line

#define EOF (-1)

The value EOF is returned by getcO when there are no more characters in the file. In C, characters have the integer value corresponding to their ASCII encoding. (See Section 3.3, "Characters and the Data Type char," on page 111.) For example, 'a' has value 97, , b' has value 98, and so forth. A char is stored in 1 byte, and an i nt is typically stored

'in either 2 or 4 bytes. Thus, a char can be considered a small integer type. Conversely, an i nt can be considered a large character type. In particular, an i nt can hold all the values of a char and other values as well, such as EOF, which is not an ordinary charac ter value. The variable c was declared to be an i nt rather than a char because it even tually would be assigned the value EOF.

II while ((c = getc(ifp)) != EOF)

if (c >= 'A' && c <= 'z')

++letter[c 'A']; /\* find uppercase letters \*/

A character is read and assigned to c. If the value of the character is not EOF, then the body of the whi 1 e loop is executed.

52 Chapter 1 T An Overview of C

• C >= 'A' && c <= 'z'

The expression c >= 'A' is true if c is greater than or equal to 'A'. Similarly, the expression c <= 'z' is true if c is less than or equal to • Z ' . The symbols && represent the logical and operator. An expression of the form

exprl && expr2

is true if and only if both exprl and expr2 are true. Because of operator precedence C >= 'A' && c <= • Z' is equivalent to (c >= 'A') && (c <= 'Z')

Thus, the expression c >= 'A' && c <= 'z' is true if and only if c has the value of an uppercase letter.

• ++letter[c - 'A'];

If c has the value 'A I, then c - I A' has the value 0. Thus, the array element 1 et ter [0] gets incremented when c has the value' A' . Similarly, if c has the value' B' , then c - 'A' has the value 1. Thus, the array element 1 ette r [ gets incremented when c has the value 'B'. In this way a count of the uppercase letters is kept in the ele ments of the array letter with letter[0] corresponding to the letter A, letter[l] corresponding to the letter B, and so forth.

• for (i = 0; i < 26; ++i) { /\* print results \*/ if (i % 6 == 0)

putc('\n', ofp);

The symbol % is the modulus operator. An expression such as a % b yields the remain der of a divided by b. For example, 5 % 3 has the value 2. In the body of the for loop we have used the expression i % 6, which has the value 0 whenever the value of i is a multiple of 6. Because of operator precedence, the expression

i % 6 == 0 is equivalent to (i % 6) o

Thus, the expression i % 6 == 0 is true every sixth time through the loop; at these times a newline character is printed. If you look at the output of the program, you will see that it is printed in six columns. The macro putcO is defined in stdio.h. It is similar to putchar 0 except that its second argument is a pointer to FILE. The value of its first argument is written to the indicated file in the format of a character.

• fprintf(ofp, "%c:%5d ", 'A' + i, letter[i]);

The function fpri ntfO is similar to pri ntfO except that it takes as its first argu

1.10 T Operating System Considerations 53 

ment a pointer to FILE. When the function is invoked, it writes to the indicated file rather than to the screen. Observe that 'A' + i is being printed in the format of a char acter. When i is 0, the expression' A' + i has the value' A', causing the letter A to be printed; when i is 1, the expression 'A' + i has the value 'B', causing the letter B to

be printed; and so forth. • Although we did not do so, we could have explicitly closed the open files just before we exited from mainO. Instead, we relied on the C system to close the files. We would use the following code to expliCitly close the files:

fclose(ifp) ;

fclose(ofp);

**1.10 Operating System Considerations**

In this section, we discuss a number of topics that are system-dependent. We begin with the mechanics of writing and running a C program.

**Writing and Running a C Program**

The precise steps that have to be followed to create a file containing C code and to com pile and execute it depend on three things: the operating system, the text editor, and the compiler. However, in all cases the general procedure is the same. We first describe in some detail how it is done in a UNIX environment. Then we discuss how it is done in an MS-DOS environment.

In the discussion that follows, we will be using the ee command to invoke the C com piler. In reality, however, the command depends on the compiler that is being used. For example, if we were using the command line version of the Turbo C compiler from Bor land, then we would use the command *tee* instead of ec. (For a list of C compilers, see the table in Section 11.13, "The C Compiler," on page 522.)

54 Chapter 1" An Overview of C 

Steps to be followed in writing and running a C program

1 Using an editor, create a text file, say *pgm.c,* that contains a C program. The name of the file must end with .c, indicating that the file contains C source code. For example, to use the vi editor on a UNIX system, we would give the command

vi pgm.c

To use an editor, the programmer must know the appropriate commands for inserting and modifying text.

2 Compile the program. This can be done with the command

cc pgm.c

The cc command invokes in turn the preprocessor, the compiler, and the loader. The preprocessor modifies a copy of the source code according to the prepro cessing directives and produces what is called a *translation unit.* The compiler translates the translation unit into object code. If there are errors, th~n the pro grammer must start again at step 1 with the editing of the source file. Errors that occur at this stage are called *syntax errors* or *compile-time errors.* If there are no errors, then the loader uses the object code produced by the compiler, along with object code obtained from various libraries provided by the system, to create the executable file *a.out.* The program is now ready to be executed.

3 Execute the program. This is done with the command

*a.out*

Typically, the program will complete execution, and a system prompt will reap pear on the screen. Any errors that occur during execution are called *run-time errors.* If for some reason the program needs to be changed, the programmer must start again at step 1.

If we compile a different program, then the file *a.out* will be overwritten, and its pre vious contents lost. If the contents of the executable file *a.out* are to be saved, then the file must be moved, or renamed. Suppose that we give the command

cc *sea.c*

This causes executable code to be written automatically into *a.out.* To save this file, we can give the command

*mv a.out sea*

1.10 " Operating System Considerations 55

causes *a.out* to be moved to *sea.* Now the program can be executed by giving the conunand

*sea*

UNIX, it is common practice to give the executable file the same name as the carre- ,sponding source file, except to drop the .c suffix. If we wish, we can use the *-0* option to direct the output of the cc command. For example, the command

ce *-0 sea sea.c*

causes the executable output from cc to be written directly into *sea,* leaving intact what ever is in *a.out.*

Different kinds of errors can occur in a program. Syntax errors are caught by the compiler, whereas run-time errors manifest themselves only during program execution. example, if an attempt to divide by zero is encoded into a program, a run-time error may occur when the program is executed. (See exercise 5, on page 61, and exercise 6, on page 61.) Usually, an error message produced by a run-time error is not very helpful in finding the trouble.

Let us now consider an MS-DOS environment. Here, some other text editor would most likely be used. Some C systems, such as Turbo C, have both a command line envi ronment and an integrated environment. The integrated environment includes both the text editor and the compiler. (Consult Turbo C manuals for details.) In both MS-DOS and UNIX, the command that invokes the C compiler depends on which C compiler is being used. In MS-DOS, the executable output produced by a C compiler is written to a file having the same name as the source file, but with the extension .exe instead of .c. Sup-

. pose, for example, that we are using the command line environment in Turbo C. If we give the command

*tee sea.c*

then the executable code will be written to *sea.exe.* To execute the program, we give the command

*sea.exe* or equivalently *sea*

To invoke the program, we do not need to type the .exe extension. If we wish to rename this file, we can use the *rename* command.

56 Chapter 1 v An Overview of C

**Interrupting a Program**

When running a program, the user may want to interrupt, or kill, the program. For example, the program may be in an infinite loop. (In an interactive environment it is not necessarily wrong to use an infinite loop in a program.) Throughout this text we assume that the user knows how to interrupt a program. In MS-DOS and in UNIX, a con

trol-c is commonly used to effect an interrupt. On some systems a special key, such as *delete* or *rubout* is used. Make sure that you know how to interrupt a program on your system.

**Typing an End-af-file Signal**

file dbLout.c

#include <stdio.h>

int main(void)

{

char Cj

while (scanf("%c", &c) \_\_ pri ntf("%c", c) j } pri ntf("%c", c);

return 0;

}

1.10 V Operating System Considerations 57 1) {

When a program is taking its input from the keyboard, it may be necessary to generate an end-of-file signal for the program to work properly. In UNIX, a carriage return fol lowed by a control-d is the typical way to effect an end-of-file signal. (See exercise 26,

we compile the program and put the executable code in the file *dbl out* th . d· t' . I h - , en, usmg re Irec lOn, we can mvo <e t e program in four ways:  *dbLout*

on page 68, for further discussion.)

**Redirection af the Input and the Output**

*dbLout < dbLout* > *dbLout* <

*infile*

*outfile*

*infile* > *outfile*

Many operating systems, including MS-DOS and UNIX, can redirect the input and the output. To understand how this works, first consider the UNIX command 

Is

This command causes a list of files and directories to be written to the screen. (The comparable command in MS-DOS is *dir.)* Now consider the command

Is > tmp

The symbol> causes the operating system to redirect the output of the command to the file *tmp.* What was written to the screen before is now written to the file *tmp.* Our next program is called *dbLout.* It can be used with redirection of both the input and the output. The program reads characters from the standard input file, which is normally connected to the keyboard, and writes each character twice to the standard output file, which is normally connected to the screen.

Used in this context, the ~ymb~ls < and> can be thought of as arrows. (See exercise 26 on page 68, for further dIscussIOn.) ,

Some commands are not meant to be used with redirection. For example, the Is com mand do~s not read characters from the keyboard. Therefore, it mal<es no sense to redi rect :he mput to the ls command; because it does not take keyboard input th . nothing to redirect. I ere IS

58 Chapter 1 V An Overview of C

**Summary** 

1 Programming is the art of communicating algorithms to computers. An algorithm is a computational procedure whose steps are completely specified and elementary.

2 The C system provides a standard library of functions that can be used by the pro grammer. Two functions from the library are pri ntfO and scanfO. They are used for output and input, respectively. The function pri ntfO can print out explicit text and can use conversion specifications that begin with the character % to print the values of the arguments that follow the control string. The use of scanfO is somewhat analogous, but conversion specifications in the control string are matched with the other arguments, all of which must be addresses (pointers). 

3 The C compiler has a preprocessor built into it. Lines that begin with a # are prepro cessing directives. A #defi ne directive can be used to define symbolic constants. A #i ncl ude directive can be used to copy the contents of a file into th~ code.

4 Statements are ordinarily executed sequentially. Special statements such as if, i f-e 1 se, for, and whi 1 e statements can alter the sequential flow of control during execution of a program.

5 A program consists of one or more functions written in one or more files. Execution begins with the function rna in O. Other functions may be called from within rna; nO and from within each other.

6 A function definition is of the form

*type (unction\_name(parameter type list)*

{

*declarations*

*statements*

}

A function definition consists of two parts, a header and a body. The header con sists of the code before the first left brace, and the body consists of the declara tions and statements between the braces.

7 In a function definition all declarations must occur before any statements. All vari ables must be declared. Compound statements are surrounded by the braces

v 5ummary 59

{ and }. Syntactically, a compound statement is itself a statement. A compound statement can be used anywhere that a simple statement can be used.

8 Although there are technical differences, macros are used like functions. The mac ros getcharO and putcharO are defined in *stdio.h.* They are used to read a char acter from the keyboard and to write a character to the screen, respectively. They are typically used by the programmer to manipulate character data.

9 A program consists of one or more functions in one or more files. Execution begins with the function rna in O. The cc command followed by a list of files that consti tutes a program creates an executable file.

10 All arguments to functions are passed call-by-value. This means that when an expression is passed as an argument, the value of the expression is computed, and it is this value that is passed to the function. Thus, when a variable is passed as an argument to a function, the value of the variable is computed, which we may think of as a copy of the variable, and it is this copy that is passed to the function. Hence, the value of the variable is not changed in the calling environment.

**11** When a return statement is encountered in a function, control is passed back to the calling environment. If an expression is present, then its value is passed back as well.

12 The use of many small functions as a programming style aids modularity and docu mentation of programs. Moreover, programs composed of many small functions are easier to debug.

13 Arrays, strings, and pointers are intimately related. A string is an array of charac ters, and an array name by itself a pointer that points to its first element. By con vention, the character \0 is used as an end-of-string sentinel. A constant string such as "abc" can be considered a pointer to type char. This string has four char acters in it, the last one being \0.

60 Chapter 1... An Overview of C

**Exercises**

1 On the screen write the words

she sells sea shells by the seashore (a) all on one line, (b) on three lines, (c) inside a box.

... Exercises 61

The following program may have a run-time error in it:

#include <stdio.h>

int main(void)

{ int x, y 0;

x 1 I y;

printf("x %d\n", x);

return 0;

}

2 Use a hand calculator to verify that the output of the marathon program is correct. Create another version of the program by changing the floating constant 1760.0 to an integer constant 1760. Compile and execute the program and notice that the out put is not the same as before. This is because integer division discards any frac tional part. 

3 Write a version of the marathon program in Section 1.3, "Variables, Expressions, and Assignment," on page 11, in which all constants and variables are of type dou b 1 e. Is the output of the program exactly the same as that of the original program?

4 Take one of your working programs and alter it by deleting the keyword voi d in the line

int main(void)

When you compile your program, does your compiler complain? ~robab~y not. (See Section 5.3, "Function Prototypes," on page 202, for further dIscussIOn.) .Next, remove the keyword vo; d and remove the following line from the body of mal nO:

return 0;

When you compile the program, does your compiler complain? This time it should. If your compiler does not complain, learn how to set a higher warnin~ level for y~ur compiler. Generally speaking, programmers should always use the highest warn:ng level possible. One of the principal rules of programming is keep your complIer happy, but not at the expense of turning off all the warnings. Programmers should rework their code repeatedly until all the warnings have vauished.

Check to see that the program compiles without any error messages. Run the pro gram to see the effect of integer division by zero. On most systems this program will exhibit a run-time error. If this happens on your system, try to rewrite the pro gram without the variable y, but keep the error in the program. That is, divide by zero directly. Now what happens?

6 Most C systems provide for logically infinite floating values. Modify the program given in the previous by changing i nt to doub 1 e and, in the p ri ntf 0 statement, %d to %f. Does the program still exhibit a run-time error? On most systems the answer is no. What happens on your system?

7 Any #i ncl ude lines in a program normally occur at the top of the file. But do they have to be at the top? Rewrite the pacific\_sea program in Section 1.4, "The Use of #defi ne and #i ncl ude," on page 15, so that the #i ncl ude line is not ilt the top of the file. For example, try

int main(void)

{ #i ncl ude "pad fi c\_sea. h"

8 Take one of your files containing a working program, say sea.c, and rename the file as sea. Now try to compile it. Some C compilers will complain; others will not. On UNIX systems, the complaint may be quite cryptic, with words such as bad magic number or unable *to* process using elf libraries. What happens on your system?

62 Chapter 1 T An Overview of C

9 The following program writes a large letter Ion the screen: #include <stdio.h>

T Exercises 63

Write a program that neatly prints a table of powers. The first few lines of the table might look like this:

.. .. .. . ~

.. .. .. ..

~ A TABLE OF POWERS

#define #define #define

BOTTOM\_SPACE HEIGHT

OFFSET

H\n\n\n\n\n" 17

II

" /,~ 17 blanks ,~/

Integer Square 3rd power 4th power 5th power ------ ---------

#define

TOP\_SPACE

"\n\n\n\n\n"

1 1

1

1

1

int main(void)

2 4 3 9

8

27

16 81

32

243

{ 

i nt i ;

printf(TOP\_SPACE);

printf(OFFSET HIIIIIII\n"); for (i = 0; i < HEIGHT; ++i) printf(OFFSET II III\n"); pri ntf(OFFSET "IIIIIII\n"); printf(BOTTOM\_SPACE);

return 0;

}

A fo r loop has the form

for *(exprl; expr2; expr3)*

*statement*

If all three expressions are present, then this is equivalent to

*exprl;*

whil e *(expr2)* {

*statement*

Compile and run this program so that you understand its effect. Write a similar program that prints a large letter C on the screen.

10 Take a working program and omit each line in turn and run it through the compiler. Record the error messages each such deletion causes. As an example, consider the folloVYing code in the file *nonsense.c:*

#include <stdio.h>

/\* forgot main \*/

*expr3;*

}

Why is there no semicolon following *statement?* Of course, it may well happen that *statement* itself contains a semicolon at the end, but it does not have to. In C, a compound statement consists of braces surrounding zero or more other state ments, and a compound statement is itself a statement. Thus, both { } and { ; ; } are statements. Try the following code:

i nt i;

{ printf("nonsense\n");

}

11 Write a program that asks interactively for your *name* and *age* and responds with He 110 *name,* next year you wi 11 be *nexLage.*

for (i { }

for (i { ;

0; i < 3; ++i) /\* no semicolon \*/ 0; i < 3; ++i)

; } /\* three semicolons, but none after the statement \*/

where *nexLage* is *age* + 1.

Is your compiler happy with this? (It should be.) Compilers care about legality. If what you write is legal but otherwise nonsense, your compiler will be happy.

14 The standard header files supplied by the C system can be found in one or more system-dependent directories. For example, on UNIX systems these header files might be in */usr/include.* On Turbo C systems they might be in *\turboc\include* or *\tc\include* or *\bc\include.* Find the location of the standard header file *sldio.h* on

64 Chapter 1" An Overview of C

your system. Read this file and find the line that pertains to pri ntfO. The line will look something like

i nt pri ntfCconst char ~'format, ... );

This line is an example of a function prototype. Function prototypes tell the com piler the number and type of arguments that are expected to be passed to the func tion and the type of the value that is returned by the function. As we will see in later chapters, strings are of type "pointer to char," which is specified by char "'. The identifier format is provided only for its mnemonic value to the programmer. The compiler disregards it. The function prototype for pri ntf 0 could just as well have been written

int printf(const char \*, ... );

The keyword const tells the compiler that the string that gets passed as an argu ment should not be changed. The ellipses ... indicate to the compiler that the number and type of the remaining arguments vary. The pri ntfO function returns as an i nt the number of characters transmitted, or a negative value if an error occurs. Recall that the first program in this chapter in Section 1.2, "Program Out put," on page 6, prints the phrase "from sea to shining C" on the screen. Rewrite the program by replacing the #i ncl ude line with the function prototype for pri ntfO given above. *Caution:* You can try to use verbatim the line that you found in *stdio.h,* but your program may fail. (See Chapter 8, "The Preprocessor.")

15 (Suggested to us by Donald Knuth at Stanford University.) In the *running\_sum* pro gram in Section 1.6, "Flow of Control," on page 26, we first computed a sum and then divided by the number of summands to compute an average. The following program illustrates a better way to compute the average:

/\* Compute a better average. \*/

#include <stdio.h>

int main(void)

" Exercises 65

for Ci = 1; scanfC"%lf", &x) 1; ++i) {

avg += Cx avg) / i;

sum += x;

navg sum / i;

printf("%5d%17e%17e%17e\n", i, x, avg, navg);

} return 0;

}

Run this program so that you understand its effects. Note that the better algorithm for computing the average is embodied in the line

avg += (x avg) / i;

Explain why this algorithm does, in fact, compute the running average. *Hint:* Do some simple hand calculations first.

In the previous exercise we used the algorithm suggested to us by Donald Knuth to write a program that computes running averages. In this exercise we want to use that program to see what happens when sum gets too large to be represented in the machine. (See Section 3.6, "The Floating Types," on page 119, for details about the values a doub 1 e can hold.) Create a say *data,* and put the following numbers into it:

le308 1 le308 1 le308

Run the program, redirecting the input so that the numbers in your file *data* get read in. Do you see the advantage of the better algorithm?

(Advanced) In this exercise you are to continue the work you did in the previous exercise. If you run the *better\_average* program taking the input from a file that contains some *ordinary* numbers, then the average and the naive average seem to be identical. Find a situation where this is not the case. That is, demonstrate experi

mentally that the better average really is better, even when sum does not overflow. 18 Experiment with the type qualifier const. How does your compiler treat the follow

{

i nt

double double double double

i ;

x;

avg

navg; sum =

0.0; 0.0;

/\* a better average \*/ /'~ a nai ve average \* /

ing code?

const int a = 0;

a 333;

printfC"%d\n", a);

pri ntf("%5s%17s%17s%17s\n%5s%17s%17s%17s\n\n", "Count", "Item", "Average", "Naive avg",

If It" II n If n

66 Chapter 1 'If An Overview of C

19 Put the folIo' wing lines into a program and run it so that you understand its effects: int a1, a2, a3, cnt; 

printf("Input three integers: ");

cnt = scanf("%d%d%d", &a1, &a2, &a3);

printf("Number of successful conversions: %d\n", cnt);

What happens if you type the letter *x* when prompted by your program? What num bers can be printed by your program? Hint: If scanfO encounters an end-of-file mark before any conversions have occurred, then the value EOF is returned, where EOF is defined in *stdio.h* as a symbolic constant, typically with the value -1. You should be able to get your program to print this number.

20 In ANSI C the pri ntfO function returns as an i nt the number of characters printed. To see how this works, write a program containing the following lines:

int cnt;

cnt printf("abc abc");

printf("\nNo. of characters printed: %d\n", cnt);

What gets printed if the control string is replaced by

"abc\nabc\n" or "abc\0abc\0"

'If Exercises 67

Use the ideas presented in the *nice\_day* program in Section 1.8, "Arrays, Strings, and Pointers," on page 39, to ""THe a program that counts the total number of let ters. Use redirection to test your program. If *infile* is a file containing text, then the command

*nletlers* < *infile*

should cause something like

Number of letters: 179

to be printed on the screen.

In the *abc* program in Section 1.8, "Arrays, Strings, and Pointers," on page 43, we used the loop

for ( ; ~'p != '\0 1 ; ++p) {

i f (~'p == 'e I)

'~p = 'E';

if (\*p == ' ')

;'p = '\n';

}

Braces are needed because the body of the for loop consists of two if statements. Change the code to

21 In the previous exercise you were able to get different numbers printed, depending on the input you provided. Put the following lines into a program:

char c1, c2, c3;

int cnt;

for C ; >"p if C"'p

"'p

else if

"'p

'\0'; ++p) Ie')

, E' ;

ep == ' ') '\n' j

printf("Input three characters: ");

cnt scanf("%c%c%c", &c1, &c2, &c3);

printf("Number of successful conversions: %d\n", cnt);

By varying the input, what numbers can you cause to be printed? Hint: The numbers printed by the program you wrote for exercise 19, on page 66, can be printed here, but you have to work much harder to do it.

Explain why braces are not needed now. Check to see that the run-time behavior of the program is the same as before. Explain why this is so.

Suppose that a is an array of some type and that i is an i nt. There is a fundamen tal equivalence between the expression a[i] and a certain corresponding pointer expression. What is the corresponding pointer expression?

Complete the following program by writing a prn\_stri ng 0 function that uses putcharO to print a string passed as an argument. Remember that strings are ter minated by the null character \0. The program uses the strcatO function from the standard library. Its function prototype is given in the header file string.h. The function takes two strings as arguments. It concatenates the two strings and puts the results in the first argument.

68 Chapter 1., An Overview of C 

#include <stdio.h>

#include <string.h>

#define MAXSTRING 100 void prn\_string(char '~)i int main(void)

**Chapter 2** 

**Lexical Elements, Operators,**

{

char sl[MAXSTRING], s2[MAXSTRING];

strcpyl(sl,' "Mary, Mary, quite contrary, \n"); strcpy(s2, "how does your garden grow?\n"); prn\_string(sl); prn\_string(s2);

strcat(sl, s2); /\* concatenate the strings \*/ prn\_string(sl);

return 0;

}

**and the C System**

In this chapter, we explain the lexical elements of the C programming language. C is a language. Like other languages, it has an alphabet and rules for putting together words and punctuation to make correct, or legal, programs. These rules are the *syntax* of the language. The program that checks on the legality of C code is called the *compiler.* If there is an error, the compiler will print an error message and stop. If there are no errors, then the source code is legal, and the compiler translates it into object code,

26 Redirection, like many new ideas, is best understood with experimentation. In Sec tion 1.10, "Operating System Considerations," on page 57, we presented the *dbLout* program. Create a file, say *myJile,* that contains a few lines of text. Try the follow ing commands so that you understand the effects of using redirection with *dbLout* 

*dbLout* < *my-file*

*dbLout* < *my-file* > *tmp*

The following command is of special interest:

*dbLout* > *tmp*

This command causes *dbLout* to take its input from the keyboard and to write its output in the file *tmp,* provided that you effect an end-of-file signal when you are finished. What happens if instead of typing a carriage return followed by a con trol-d, you type a control-c to kill the program? Does anything at all get written into *tmp?*

which in turn gets used by the loader to produce an executable file. When the compiler is invoked, the preprocessor does its work first. For that reason we can think of the preprocessor as being built into the compiler. On some systems, this is actually the case, whereas on others the preprocessor is separate. This is not of concern to us in this chapter. We have to be aware, however, that we can get error mes sages from the preprocessor as well as from the compiler. (See exercise 30, on page 106.) Throughout this chapter, we use the term *compiler* in the sense that, conceptually, the preprocessor is built into the compiler. 

A C program is a sequence of characters that will be converted by a C compiler to object code, which in turn gets converted to a target language on a particular machine. On most systems, the target language will be a form of machine language that can be run or interpreted. For this to happen, the program must be syntactically correct. The compiler first collects the characters of the program into *tokens,* which can be thought of as the basic vocabulary of the language.

In ANSI C, there are six kinds of tokens: keywords, identifiers, constants, string con stants, operators, and punctuators. The compiler checks that the tokens can be formed into legal strings according to the syntax of the language. Most compilers are very pre cise in their requirements. Unlike human readers of English, who are able to understand the meaning of a sentence with an extra punctuation mark or a misspelled word, a C compiler will fail to provide a translation of a syntactically incorrect program, no mat ter how trivial the error. Hence, the programmer must learn to be precise in writing code.

l

70 Chapter 2 'f Lexical Elements, Operators, and the C System

The programmer should strive to v\Tite understandable code. A key part of doing this is producing well-commented code with meaningful identifier names. In this chapter we illustrate these important concepts.

The compilation process

~

2.1 'f Characters and Lexical Elements

71

1\* Read in two integers and print their sum. \*1

#include <stdio.h>

~ Group characters ----I....... into tokens

**2.1 Characters and Lexical Elements**

Translate tokens to target code

int a, b, sum;

printfCHlnput two integers: If); scanfC"%d%d", &a, &b);

sum = a + b;

printfC"%d + %d = %d\n", a, b, sum); return 0;

A C program is first constructed by the programmer as a sequence of characters. Among the characters that can be used in a program are the following:

Characters that can be used in a program

lowercase letters a b c z

uppercase letters A B C Z

digits 0 1 2 3 4 5 6 7 8 9

other characters + *i'I=C)* { } [ ] < > I If

# % & - I A \_ \ , ; ?

white space characters blank, newline, tab, etc.

These characters are collected by the compiler into syntactic units called *tokens.* Let us look at a simple program and informally pick out some of its tokens before we go on to a strict definition of C syntax.



>cical Dissection of the *sum* Program

*I'll* Read in two integers and pri nt thei r sum. \*1

;Olllillents are delimited by r and \*1. The compiler first replaces each comment by a blank. Thereafter, the compiler either disregards white space or uses it to sepa tokens.

<stdio.h>

is a preprocessing directive that causes the standard header file *stdio.h* to be .IIlClUuea. We have included it because it contains the function prototypes for pr; ntf() scanfO. A function prototype is a lund of declaration. The compiler needs func prototypes to do its work.

int main(void)

{ int a, b, sum;

compiler groups these characters into four kinds of tokens. The function name n is an identifier, and the parentheses ( ) immediately follOwing mai n is an opera- . This idea is confusing at first, because what you see following ma; n is (vo; d), but IS only the parentheses ( ) themselves that constitute the operator. This operator the compiler that ma; n is a function. The characters "{", ", ", and";" are punctua i nt is a keyword; a, b, and sum are identifiers.

72 Chapter 2.., Lexical Elements, Operators, and the C System

II i nt a, b, sum;

The compiler uses the white space between i nt and a to distinguish the two tokens. We cannot write

inta, b, sum; /\* wrong: white space is necessary \*/

On the other hand, the white space following a comma is superfluous. We could have written

i nt a,b,sum; but not int absum;

The compiler would consider absum to be an identifier.

2.2 .., Syntax Rules

73

a + bj

acters = and + are operators. White space here will be ignored, so we could written

or sum a + b

u m a + b;

'cc,;;;,"",-;,,, had written the latter, then each letter on this line would be treated by the com as a separate identifier. Because not all of these identifiers have been declared, the

II printfC"Input two integers: ");

scanfC"%d%d" , &a, &b);

The names pri ntf and scanf are identifiers, and the parentheses following them tell the compiler that they are functions. After the compiler has translated the C code, the loader will attempt to create an executable file. If the code for pri ntfO and scanfO has not been supplied by the programmer, it vvill be taken from the standard library. A programmer would not normally redefine these identifiers.

II "Input two integers: "

A series of characters enclosed in double quotes is a string constant. The compiler treats this as a single token. The compiler also provides space in memory to store the string.

II &a, &b

The character & is the address operator. The compiler treats it as a token. Even though the characters & and a are adjacent to each other, the compiler treats each of them as a separate token. We could have written

compiler either ignores white space or uses it to separate elements of the lan-• 

.tc;v.'UfJU'-J would complain.

.. "" .. "'''''''. The programmer uses white space to provide more legible code. To the compiler, nr()gyiam text is implicitly a single stream of characters, but to the human reader, it is a WU-UIUlt::ll;:'lVll<U tableau.

**Syntax Ru les**

The syntax of C will be described using a rule system derived from Backus-Naur Form }(~NF), first used in 1960 to describe ALGOL 60. Although they are not adequate by themselves to describe the legal strings of C, in conjunction with some explanatory they are a standard form of describing modern high-level languages.

A syntactic category will be written in italics and defined by productions, also called ',-"'.LLlJlH'" rules, such as

&

but not &a &b

a & b or &a,&b

/\* the comma punctuator is missing \*/

*digit* :: = 0 I 1 I 2 I 3 I 4 I 5 I 6 I 7 I 8 I 9

The syntactic category *digitis* rewritten as either the symbol 0, the symbol 1, ... , or

a&, &b

/\* & requires its operand to be on the right \*/

the symbol 9.

The vertical bar separates alteruate choices. Symbols not in italics are taken to be termi nal symbols of the language to which no further productions are applied.



74 Chapter 2 v Lexical Elements, Operators, and the C System

Symbols to be used in productions

| *italics*  , ~~.-~~  , ~~,-~~ | indicate syntactic categories  "to be rewritten as" symbol |
| --- | --- |
| I | vertical bar to separate choices |
| { h | choose 1 of the enclosed items |
| { }0+ | repeat the enclosed it~ms 0 or more times |
| { h+ | repeat the enclosed items 1 or more times |
| { }opt | optional items |

!

Other items are terminal symbols of the language,

2.3 V Comments

75

*a\_string* :: = *uppercase\_letter alphanumericstring*

define a syntactic category called *conditionaLstatement* to illustrate the { }opt as follows:

: : = i f *(expression) statement*

{e 1 se *statement}* opt

*expression* and *statement* have not yet been supplied with rewriting rules, this is not defined completely. Those rewriting rules are complicated, and we are to present them here. In any case, some examples of this syntactic category

f (big\_big\_big > 999)

Let us define a category *letter\_or\_digit* to mean any lower- or uppercase letter of the alphabet or any decimal digit, Here is one way we can do this:

*letter\_or\_digit* :: *letter* I *digit*

*letter* :: = *lowercase\_letter* I *uppercase\_letter*

*lowercase\_letter* :: = a I b I c I .. , I z

*uppercase\_letter* :: = A I B I C I ... I Z

*digit* :: 0 I 1 I 2 I 3 I 4 I 5 , 6 , 7 I 8 I 9

Now let us create a category *alphanumeric\_string* to be an arbitrary sequence of letters or digits.

*alphanumericstring* :: = *{letter\_or\_digit}0+*

Using these productions, we see that strings of one character such as "3" and strings of many characters such as "ab777c" as well as the null string '"' are all alphanumeric strings. Note that in each of our examples double-quote characters were used to delimit the alphanumeric string. The double-quote characters themselves are not part of the string.

If we wish to guarantee that a string has at least one character, we must define a new syntactic category, such as

*aJpha\_string\_l* :: *{letter\_or\_digith+*

and if we want strings that start with an uppercase letter, we could define

huge = giant + ot; /\* no else part immediately follows \*/

(normalized\_score >= 65)

pass 1;

lse /\* else part associated with preceding if part \*/ pass = 0;

**Comments**

1l1l.l.lC.llli:> are arbitrary strings of symbols placed between the delimiters /\* and 1(/ ' are not tokens. The compiler changes each comment into a single blank . Thus, comments are not part of the executable program. We have already examples such as

a comment 1(/ /\*\*\* another comment \*\*\*/ example is

A comment can be written in this fashion

to set it off from the surrounding code.

76 Chapter 2" Lexical Elements, Operators, and the C System 

The following illustrates one of many styles that can be used to highlight comments:

*1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

\* If you wish, you can \*

\* put comments in a box. \*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*1*

Comments are used by the programmer as a documentation aid. The aim of docu mentation is to explain clearly how the program works and how it is to be used. Some times a comment contains an informal argument demonstrating the correctness of the program.

Comments should be written simultaneously with program text. Although some pro grammers insert comments as a last step, there are two problems with this approach. The first is that once the program is running, the tendency is to either omit or abbrevi ate the comments. The second is that ideally the comments should serve as running commentary, indicating program structure and contributing to program clarity and cor rectness. They cannot do this if they are inserted after the coding is finished.

In C++, the C style of comment is still valid. But in addition, C++ provides for com ments that begin with *I I* and run to the end of the line.

*II* This is a comment in c++.

*II II* This is one common way of writing *II* a comment in C++ that consists

*II* of many lines.

*II*

1"(

*II* This C comment style mimics the

*II* previous C++ comment style.

~(I

2.4 " Keywords

**Keywords**

T{/lvWOf(lS are explicitly reserved words that have a strict meaning as individual tokens They cannot be redefined or used in other contexts.

Keywords

auto do goto signed unsigned

break double if sizeof void

case else int static volatile

char enum long struct while

const extern register switch

continue float return typedef

default for short union

implementations may have additional keywords. These will vary from one imple tion, or system, to another. As an example, here are some of the additional key words in Turbo C.

Additional keywords for Borland C

asm cdecl far huge interrupt near pascal

Compared to other major languages, C has only a small number of keywords. Ada, for e'Xample, has 62 keywords. It is a characteristic of C that it does a lot with relatively few special symbols and keywords.

77

',::

78 Chapter 2.., Lexical Elements, Operators, and the C System

**2.5 Identifiers**

An identifier is a token that is composed of a sequence of letters, digits, and the special character \_, which is called an *underscore.* A letter or underscore must be the first char acter of an identifier. In most implementations of C, the lower- and uppercase letters are treated as distinct. It is good programming practice to choose identifiers that have mnemonic significance so that they contribute to the readability and documentation of the program.

*identifier {letter* I *underscoreh {letter* I *underscore* I *digit}0+ underscore* :: = \_

Some examples of identifiers are

k

d

iamanidentifier2

so\_am\_i

but not

2.6 .., Constants 

79

programming style requires the programmer to choose names that are mean If you were to write a program to figure out various taxes, you might have iden such as tax\_rate, pri ee, and tax, so that the statement

have an obvious meaning. The underscore is used to create a single identifier what would normally be a string of words separated by spaces. Meaningfulness avoiding confusion go hand in hand with readability to constitute the main guide for a good programming style.

"n"ru'YI'Identifiers that begin with an underscore can conflict with system names, systems programmers should use such identifiers, Consider the identifier ob, is often defined as the name of an array of structures in *stdio.h.* If a programmer to use \_ ; ob for some other purpose, the compiler may complain, or the program misbehave. Applications programmers are advised to use identifiers that do not

with an underscore. Also, external identifiers can be subjected to system-depen restrictions,

**Constants**

not#me 

10Lsouth -plus

/\* special character # not allowed \*/ /\* must not start with a digit \*/ /\* do not mistake - for \_ \*/

we have seen in some simple introductory programs, C manipulates various kinds of Numbers such as 0 and 17 are examples of integer constants, and numbers such

Identifiers are created to give unique names to objects in a program. Keywords can be thought of as identifiers that are reserved to have special meaning. Identifiers such as scanf and pri ntf are already known to the C system as input/output functions in the standard library. These names would not normally be redefined. The identifier ma; n is special, in that C programs always begin execution at the function called mai n.

One major difference among operating systems and C compilers is the length of dis criminated identifiers. On some older systems, an identifier with more than 8 charac ters will be accepted, but only the first 8 characters will be used. The remaining characters are simply disregarded. On such a system, for example, the variable names

and 

would be considered the same.

In ANSI C, at least the first 31 characters of an identifier are discriminated. Many C systems discriminate more.

1.0 and 3.14159 are examples of floating constants. Like most languages, C treats and floating constants differently. In Chapter 3, "The Fundamental Data Types," will discuss in detail how C understands numbers. There are also character con in C, such as ! a " 'b I , and '+'. Character constants are written between single and they are closely related to integers. Some character constants are special, as the newline character, written "\n". The backslash is the escape character, and think of \n as "escaping the usual meaning of n." Even though \n is wTitten ''lith the 'i",i'UVV characters \ and n, it represents a single character called *newline.* addition to the constants that we have already discussed, there are enumeration ('f\nQ!',:n,rc in C. We will discuss these along with the keyword enum in Section 7.5, "Enu Types," on page 345. Integer constants, floating constants, character con and enumeration constants are all collected by the compiler as tokens. Because implementation limits, constants that are syntactically expressible may not be avail on a particular machine. For example, an integer may be too large to be stored in a machine word.



80 Chapter 2" Lexical Elements, Operators, and the C System

Decimal integers are finite strings of decimal digits. Because C provides octal and hexadecimal integers as well as decimal integers, we have to be careful to distinguish between the different kinds of integers. For example, 17 is a decimal integer constant, 017 is an octal integer constant, and 0x17 is a hexadecimal integer constant. (See Chap

of text"

" I~' a stri ng H == b + c;" I~' a .' '1" /\* th1S 1S not a comment H

2.8 " Operators and Pu nctuators

*1\** the **null** string *\*1*

of blank characters *\*1*

nothing is executed *\*1*

81

ter 3, "The Fundamental Data Types," for further discussion.) Also, negative constant integers such as -33 are considered constant expressions.

*decimaUnteger* :: = 0 I *positive\_decimaUnteger*

*positive\_decimaUnteger* :: = *positive\_digit {digitJe+*

*positive\_digit* :: = 1 I 2 I 3 I 4 I 5 I 6 I 7 I 8 I 9

Some examples of constant decimal integers are

o

"a string with double quotes \" within"

!fa single backslash \\ is **in** this string"

*r* "this is not a string" ~'I

"and

neither is this"

sequences that would have meaning if outside a string constant are just a

77

123456789000 but not

0123

-49

123.0

*1\** too large for the machine? *\*1*

*1\** an octal integer *\*1*

*1\** a constant expression *\*1 1\** a floating constant *\*1*

1!t:1./""''-H'-~ of characters when surrounded by double quotes. In the previous examples, string contains what appears to be the statement a = b + C;, but since it occurs "<I111'rOllnClea by double quotes, it is explicitly this sequence of characters. Two string constants that are separated only by white space are concatenated by the ,.\;VJlUVU~L into a single string. Thus,

is equivalent to "abcdef"

Although we have already used integer constants such as 144 and floating constants such as 39.7, their meaning in terms of type, along with details concerning memory requirements and machine accuracy, is complicated enough to require a thorough dis cussion. We do this in Chapter 3, "The Fundamental Data Types."

**2.7 String Constants** 

A sequence of characters enclosed in a pair of double-quote marks, such as "abc", is a string constant, or a string literal. It is collected by the compiler as a single token. In Section 6.10, "Strings," on page 270, we will see that string constants are stored by the compiler as arrays of characters. String constants are always treated differently from character constants. For example, 'a' and "a" are not the same.

Note that a double-quote mark" is just one character, not two. If the character" itself is to occur in a string constant, it must be preceded by a backslash character \ . If the character \ is to occur in a string constant, it too must be preceded by a backslash. Some examples of string constants are

is a new feature of the language available in ANSI C, but not in traditional C. constants are treated by the compiler as tokens. As with other constants, the *!-<V'U/-"CL'-L* provides the space in memory to store string constants. We will emphasize point again in Section 6.10, "Strings," on page 270, when we discuss strings and

**Operators and Punctuators**

C, there are many special characters with particular meanings. Examples include the C\rithmetic operators

+ I %

which stand for the usual arithmetic operations of addition, subtraction, multiplication, diviSion, and modulus, respectively. Recall that in mathematics the value of *a* modulus

i ~ i 82 Chapter 2... lexical Elements, Operators, and the C System 

*b* is obtained by taking the remainder after dividing *a* by *b.* Thus, for example, 5 % 3 has the value 2, and 7 % 2 has the value 1.

In a program, operators can be used to separate identifiers. Although typically we put white space around binary operators to heighten readability, this is not required.

/\* this is the expression a plus b \*/

/\* this is a 3-character identifier \*/

Some symbols have meanings that depend on context. As an example of this, consider the % symbol in the two statements

pri ntf("%d", a); and a b % 7;

The first % symbol is the start of a conversion specification, or format, whereas the sec ond % symbol represents the modulus operator.

Examples of punctuators include parentheses, braces, commas, and semicolons. Con sider the following code:

int main(void)

2.9 ... Precedence and Associativity of Operators

83

**Precedence and Associativity of Operators**

~"'~~<"-rlr' have rules of *precedence* and *associativity* that are used to determine how cc;.~,vnreS:SllJ11" are evaluated. These rules do not fully determine evaluation because the has leeway to make changes for its own purposes. Since expressions inside ses are evaluated first, parentheses can be used to clarify or change the order operations are performed. Consider the expression

the operator'" has higher precedence than +, causing the multiplication to be per- ""',wl"n£.n first, then the addition. Hence, the value of the expression is 7. An equivalent

3)

{

i nt a, b = 2, c a = 17 \* (b + c);

other hand, because expressions inside parentheses are evaluated first, the

3 ;

The parentheses immediately following mai n are treated as an operator. They tell the compiler that ma; n is the name of a function. After this, the symbols "{", ",", ";", U(", and ")" are punctuators. 

Both operators and punctuators are collected by the compiler as tokens, and along with white space, they serve to separate language elements.

Some special characters are used in many different contexts, and the context itself can determine which use is intended. For example, parentheses are sometimes used to indicate a function name; at other times they are used as punctuators. Another example is given by the expressions

a + b ++a a += b

They all use + as a character, but ++ is a single operator, as is +=. Having the meaning of a symbol depend on context makes for a small symbol set and a terse language.

isccUfterent; its value is 9. Now consider the expression

- 3 + 4 5

e the binary operators + and have the same precedence, the associativity rule to right" is used to determine how it is evaluated. The "left to right" rule means operations are performed from left to right. Thus,

5

eqUivalent expression.

following table gives the rules of precedence and associativity for some of the periato:rs of C. In addition to the operators we have already seen, the table includes that will be discussed later in this chapter.

84 Chapter 2 T Lexical Elements, Operators, and the C System

Operator precedence and associativity

Operator Associativity

o ++ (postfix) -- (postfix) left to right + (unary) (unary) ++ (prefix) -- (prefix) right to left / % left to right 

+ left to right  etc. right to left

All the operators on a given line, such as '~, *I,* and %, have equal precedence with respect to each other, but have higher precedence than all the operators that occur on the lines below them. The associativity rule for all the operators on a given line appears at the right side of the table. Whenever we introduce new operators, we will give their rules of precedence and associativity, and often we will encapsulate the information by augmenting this table. These rules are essential information for every C programmer.

In addition to the binary plus, which represents addition, there is a unary plus, and both these operators are represented by a plus sign. The minus sign also has binary and unary meanings. Note carefully that the unary plus was introduced with ANSI C. There is no unary plus in traditional C, only unary minus.

From the preceding table we see that the unary operators have higher precedence than the binary plus and minus. In the expression

-a"'b-c

the first minus sign is unary and the second binary. Using the rules of precedence, we see that the following is an equivalent expression:

((- a) '" b) - c

85

2.10 T Increment and Decrement Operators

**Increment and Decrement Operators**

n¥DTI1""nt operator ++ and the decrement operator -- are unary operators. They IIlULlU."" because they can be used as both prefix and postfix operators. Suppose a variable of type i nt. Then both ++va 1 and val ++ are valid expressions, with illustrating the use of ++ as a prefix operator and val ++ illustrating the use of ++ operator. In traditional C, these operators have the same precedence as the

unary operators. In ANSI C, for technical reasons they have the very highest pre and left-to-right associativity as postfix operators, and they have the same pre as the other unary operators and right-to-Ieft associativity as prefix operators.

++ and -- can be applied to variables, but not to constants or ordinary expres Moreover, different effects may occur depending on whether the operators occur or postfix position. Some examples are

/\* constants cannot be incremented \*1

b - 1) 1\* ordinary expressions cannot be incremented "'/

of the expressions ++i and i ++ has a value; moreover, each causes the stored of i in memory to be incremented by 1. The expression ++ i causes the stored of i to be incremented first, with the expression then taking as its value the new

value of i. In contrast, the expression i ++ has as its value the current value of i; the expression causes the stored value of i to be incremented. The following code the situation:

a, b, c 0;

++c;

%d %d\n", a, b, ++c); 1\* 1 1 3 is printed \*1

86 Chapter 2 T Lexical Elements, Operators, and the C System

In a similar fashion, the expression --i causes the stored value of i in memory to be decremented by 1 first, with the expression then taking this new stored value as its value. With i --, the value of the expression is the current value of i; then the expres

sion causes the stored value of i in memory to be decremented by l. Note carefully that ++ and cause the value of a variable in memory to be changed. Other operators do not do this. For example, an expression such as a + b leaves the values of the variables a and b unchanged. These ideas are expressed by saying that the operators ++ and -- have a *side effect,* not only do these operators yield a value, they also change the stored value of a variable in memory. (See exercise 14, on page 99.) In some cases, we can use ++ in either prefix or postfix position, with both uses pro ducing equivalent results. For example, each of the two statements

++i; and i++;

is eqUivalent to

i i + **1;**

In simple situations, one can consider ++ and -- as operators that provide concise nota tion for the incrementing and decrementing of a variable. In other situations, careful attention must be paid as to whether prefix or postfix position is desired.

Declarations and initializations

int a = 1, b = 2, c = 3, d '" 4;

Expression Equivalent expression Value

a'" b / c (a ,., b) / c 0

| a *it* b % c + 1 | ((a \* b) % c) + 1 | 3 |
| --- | --- | --- |
| ++a'''b-c -- | ((++ a) ," b) (c .) | 1 |
| *7--b i*'++ d | 7 - ((-b) \* (++ d)) | 17 |

2.11 T Assignment Operators 

87

**Assignment Operators**

the value of a variable, we have already made use of assignment statements

b + c;

other languages, C treats as an operator. Its precedence is lower than all the we have discussed so far, and its associativity is right to left. In this section in detail the significance of this.

understand = as an operator, let us first consider + for the sake of comparison. operator + takes two operands, as in the expression a + b. The value of the sian is the sum of the values of a and b. By comparison, a simple assignment '.L'-U'V~~~~ is of the form

*righLside*

*righLside* is itself an expression. Notice that a semicolon placed at the end would made this an assignment statement. The assignment operator = has the two oper *variable* and *righLside.* The value of *righLside* is assigned to *variable,* and that

becomes the value of the assignment expression as a whole. To illustrate this, C!onSllCler the statements

2 ;

3;

b + c;

the variables are all of type i nt. By making use of assignment expressions, we condense this to

2) + (c = 3);

assignment expression b = 2 assigns the value 2 to the variable b, and the assign expression itself takes on this value. Similarly, the assignment expression c = 3 the value 3 to the variable c, and the assignment expression itself takes on this Finally, the values of the two assignment expressions are added, and the result is assigned to a.

88 Chapter 2 'f Lexical Elements, Operators, and the C System

Although this example is artificial, there are many situations where assignment occurs naturally as part of an expression. A frequently occurring situation is multiple assignment. Consider the statement

a = b = c = 0;

Because the operator = associates from right to left, an equivalent statement is a = (b = (c = 0));

First, c is assigned the value 0, and the expression c = 0 has value 0. Then b is assigned the value 0, and the expression b (c = 0) has value 0. Finally, a is assigned the value 0, and the expression a = (b = (c = 0)) has value 0. Many languages do not use assignment in such an elaborate way. In this respect C is different.

2.12 'f An Example: Computing Powers of 2

89

+ 3 is equivalent to j = j ,~ (k + 3)

k + 3

table illustrates how assignment expressions are evaluated:

ons and initializations

1, j = 2, k = 3, m = 4;

Equivalent expression Equivalent expression Value + k i += (j + k) i (; + (j + k)) 6

In addition to =, there are other assignment operators, such as += and -= . An expres

5

sion such as



will add 2 to the old value of k and assign the result to k, and the expression as a whole

| j \*= (k = em + 5)) | j (j ,', (k (m + 5))) | 18 |
| --- | --- | --- |

will have that value. The expression

k += 2

accomplishes the same task The following table contains all the aSSignment operators:

Assignment operators

+= ~'= /= %= »= «= &"" A= 1=

All these operators have the same precedence, and they all have right-to-Ieft associativ ity. The semantics is specified by

variable OP=' expression

which is equivalent to

variable = variable op (expression)

with the exception that if variable is itself an expression, it is evaluated only once. When dealing with arrays, this is an important technical point. Note carefully that an assignment expression such as

**An Example: Computing Powers of 2**

some of the ideas presented in this chapter, we will write a program that a line some powers of 2. Here is the program:

Some powers of 2 are printed. \*/

main(void)

i = 0, power l' ,

while (++i <= 10)

pri ntf("%-6d" , power **''J'(=** 2);

printf("\n");

return 0;

t of the program is

4 8 16 32 64 128 256 512 1024

**...**

90 Chapter 2"" Lexical Elements, Operators, and the C System

• • Dissection of the *pow\_of\_2* Program

• /\* Some powers of 2 are printed. \*/

Programs often begin with a comment explaining the program's intent or use. In a large program, comments may be extensive. The compiler treats comments as white space.

• #include <stdio.h>

The header file *stdio.h* contains the function prototype for the pri ntfO function. This is a kind of declaration for pri ntfO. The compiler needs it to do its work correctly. (See Section 2.13, "The C System," on page 91, for further details.)

• int i = 0, power l' ,

The variables i and powe r are declared to be of type i nt. They are initialized to 0 and 1, respectively.

• while (++i <= 10)

As long as the value of the expression ++i is less than or equal to 10, the body of the whi 1 e loop is executed. The first time through the loop, the expression ++i has the value 1; the second time through the loop, ++i has the value 2; and so forth. Thus, the body of the loop is executed ten imes.

• pri ntfC"%-6d", power '1'= 2);

The body of the whi 1 e loop consists of this statement. The string constant "%-6d" is passed as the first argument to the pri ntfO function. The string contains the format %-6d, which indicates that the value of the expression power '1'= 2 is to be printed as a decimal integer with field width 6. The minus sign indicates that the value is to be left adjusted in its field.

• power *i'=* 2

This assignment expression is equivalent to power == power \* 2 which causes the old value of power to be multiplied by 2 and the resulting value to be assigned to power. The value assigned to powe r is the value of the assignment expression as a whole.

2.13 " The C System

91

first time through the whi 1 e loop, the old value of powe r is 1, and the new value is second time through the loop, the old value of power is 2, and the new value is 4, so forth.

**The C System**

C system consists of the C language, the preprocessor, the compiler, the library, other tools useful to the programmer, such as editors and debuggers. In this sec we discuss the preprocessor and the library. (See Chapter 8, "The Preprocessor,"

further details about the preprocessor. See Appendix A, "The Standard Library," for about functions in the standard library.) .

**Preprocessor**

that begin with a # are called *preprocessing directives.* These lines communicate the preprocessor. In traditional C, preproceSSing directives were required to begin 1. In ANSI C, this restriction has been removed. Although a # may be pre on a line by white space, it is still a common programming style to start prepro directives in column 1.

have already used preprocessing directives such as

and #define PI 3.14159

form of the #i ncl ude facility is given by

ncl ude *"filename"*

causes the preprocessor to replace the line with a copy of the contents of the file. A search for the file is made first in the current directory and then in other -dependent places. With a preprocessing directive of the form

preproc~ssor looks for the file only in the "other places" and not in the current

#i ncl ude directives commonly occur at the beginning of the program, the files they refer to are called *header files,* and a . h is used to end the file name.

92 Chapter 2 v Lexical Elements, Operators, and the C System

This is a convention; the preprocessor does not require this. There is no restriction on what an include file can contain. In particular, it can contain other preprocessing direc tives that will be expanded by the preprocessor in turn. Although files of any type may be included, it is considered poor programming style to include files that contain the code for function definitions. (See Chapter 5, "Functions.")

On UNIX systems, the standard header files such as *stdio.h* are typically found in the directory */usr/include.* On Borland C systems, they might be found in the directory *c:\bc\include* or in some other directory. In general, the location of the standard #i ncl ude files is system-dependent. All of these files are readable, and programmers, for a variety of reasons, have occasion to read them.

One of the primary uses of header files is to provide function prototypes. For exam ple, the file *stdio.h* contains the following lines:

int printf(const char \*format, ... );

93

2.13 V The C System

*to provide the (unction prototype.* This is usually accomplished by including te header files.

*tion:* Do not mistake header files for the libraries themselves. The standard contains object code of functions that have already been compiled. The stan files do not contain compiled code.

an illustration of the use of a function in the standard library, let us show how o can be used to generate some randomly distributed integers. In later chapters have occasion to use rand 0 to fiH arrays and strings for testing purposes. use it to print some integers on the screen.

nclude <stdio.h>

int

scanf(const char \*format, ... );

nclude <stdlib.h>

These are the function prototypes for the pri ntfO and scanfO functions in the stan dard library. Roughly spealdng, a function prototype tells the compiler the types of the arguments that get passed to the function and the type of the value that gets returned by the function. Before we can understand the function prototypes for pri ntfO and scanfO, we need to learn about the function-definition mechanism, pointers, and type qualifiers. These ideas are presented in later chapters. The point we are maldng here is that when the programmer uses a function from the standard library, then the corre sponding standard header file should be included. The header file will provide the appropriate function prototype and other necessary constructs. The compiler needs the function prototype to do its work correctly.

**The Standard library**

i nt i, n;

printf("\n%s\n%s",

"Some randomly distributed integers "How many do you want to see? "); scanf("%d", &n) j

for (i = 0; i < n; ++i) {

if (i % 10 0)

putchar (' \n!) j

pri ntf("%7d", rand 0) ;

} printf("\n\n");

return 0;

will be printed.",

The standard library contains many useful functions that add considerable power and flexibility to the C system. Many of the functions are used extensively by all C program mers, whereas other functions are used more selectively. Most programmers become acquainted with functions in the standard library on a need-to-know basis.

e that we execute the program and type 19 when prompted. Here is what on the screen:

randomly distributed integers will be printed.

many do you want to see? 23

Programmers are not usually concerned about the location on the system of the stan dard library because it contains compiled code that is unreadable to humans. The stan dard library may comprise more than one file. The mathematics library, for example, is conceptually part of the standard library, but it often exists in a separate file. (See exer cise 25, on page 105, and exercise 26, on page 105.) Whatever the case, the system knows where to find the code that corresponds to functions from the standard library, such as pri ntfO and scanfO, that the programmer has used. Note carefully, how ever, that even though the system provides the code, it *is the responsibility of the pro*

16838 16212 25089

5758 4086 21183

10113 2749 25137

17515 12767 25566

31051 9084 26966

5627 12060 4978

23010 32225 20495

7419 17543

94 Chapter 2 T Lexical Elements, Operators, and the C System • • 

Dissection of the prn\_rand Program

• #include <stdio.h>

#include <stdlib.h>

These header files are included because of the function prototypes they contain. In par ticular, the function prototype

int rand(void);

is in *stdlib.h.* It tells the compiler that rand 0 is a function that takes no arguments and returns an i nt value. Rather than include *stdlib.h,* we could instead supply this line our selves at the top of the file just before main O. However, it is both easier and safer to include the header file.

• printf("\n%s\n%s" , "Some randomly distributed integers will be printed. H, "How many do you want to see? H);

scanf("%d", &n);

A prompt to the user is printed on the screen. The characters typed in by the user are received by scanfO, converted in the format of a decimal integer, and placed at the address of n.

95

2.1 3 T The C System

way to write this program would be to initialize i to zero and then use the con- < n) {

that i++ < n is different from ++i < n. (See exercise 19, on page 101.)

(; % 10 == 0)

putchar('\n');

rintf("%7d H, randO);

tor == is the "is equal to" operator. If *exprl* and *expr2* are two expressions the same value, then the expression *exprl* == *expr2* will be *true;* otherwise it will In Section 4.3, "Equality Operators and Expressions," on page 152, we will see has lower precedence than % . Thus,

o is equivalent to (i % 10)

time through the loop and every eighth time thereafter, the expression as a *true.* Whenever the expression is *true,* a newline character gets printed.

ntf(H%7d", randO);

time through the loop, the value returned by the call to randO is printed in the

• for (i = 0; ; < n; ++i) { }

This is a for loop. It is equivalent to

i == 0;

while (i < n) {

++i;

}

• •

of a decimal integer. The width of the field where the integer gets printed is 7.

uses of rand 0, the programmer needs to seed the random-number generator it gets used. This can be done v\lith the following line:

exerc:ise 22, on page 102, for further discussion. Also, see exercise 25, on page 105, use of other random-number generators.)

**~~-~~**

96 Chapter 2" Lexical Elements, Operators, and the C System

**Summary**

1 Tokens are the basic syntactic units of C. They include keywords, identifiers, con stants, string constants, operators, and punctuators. White space, along with opera tors and punctuators, can serve to separate tokens. For this reason, white space, operators, and punctuators are collectively called separators. White space, other than serving to separate tokens, is ignored by the compiler.

2 Comments are enclosed by the bracket pair /'/< and "'1 and are treated as white space by the compiler. They are critical for good program documentation. Com ments should assist the reader to both use and understand the program.

3 A keyword, also called a reserved word, has a strict meaning. There are 32 key words in C. They cannot be redefined.

4 Identifiers are tokens that the programmer uses chiefly to name variables and func tions. They begin with a letter or underscore and are chosen to be meaningful to the human reader.

5 Some identifiers are already known to the system because they are the names of . functions in the standard library. These include the input/output functions scanfO and pri ntfO and mathematicalfunctions such as sqrtO, si nO, cosO, and tanO.

6 Constants include various kinds of integer and floating constants, character con stants such as I a' and '#', and string constants such as "abc". All constants are collected by the compiler as tokens.

7 String constants such as "deep bl ue seal! are arbitrary sequences of charact~rs, including white-space characters, that are placed inside double quotes. A strmg constant is stored as an array of characters, but it is collected by the compiler as a single token. The compiler provides the space in memory needed to store a string constant. Character constants and string constants are treated differently. For example, 'x' and II x 11 are not the same.

" Summary

and punctuators are numerous in C. The parentheses that follow rna; n in

an operator; they tell the compiler that rna; n is a function. The parenthe in the expression a \* (b + c) are punctuators. The operations inside the )(ll'-U",,~ses are done first.

C, the rules of precedence and associativity for operators determine how an gets evaluated. The programmer needs to know them.

increment operator ++ and the decrement operator -- have a side effect. In to having a value, an expression such as ++i causes the stored value of i memory to be incremented by 1.

operators ++ and can be used in both prefix and postfix positions, possibly different effects. The expression ++; causes i to be incremented in memory, the new value of ; is the value of the expression. The expression; ++ has as its the current value of ; , and then the expression causes; to be incremented in

C, the assignment symbol is an operator. An expression such as a = b + c UC>C>"b'U'> the value of b + c to a, and the expression as a whole takes on this value. uu.<V ..... F,H the assignment operator in C and the equal sign in mathematics look they are not comparable.

standard library contains many useful functions. If the programmer uses a from the standard library, then the corresponding standard header file be included. The standard header file provides the appropriate function

97

,.J........

98 Chapter 2.... Lexical Elements, Operators, and the C System

**Exercises**

1 Is mai n a keyword? Explain.

2 List five keywords and explain their use.

3 Give two examples of each of the six types of tokens. 4 Which of the following are not identifiers and why?

99

.... Exercises

+ b or a + ++b

lC:UU.U"EJ on how the plus symbols are grouped. The correct grouping is the first one. This is because the compiler groups the longest string as a token first, and so uses ++ instead of + as a first token. Write a short program to check this.

For the pow\_of\_2 program in Section 2.12, "An Example: Computing Powers of 2," on page 89, ~xplain what the effect would be if the expression ++i were changed to

The following code can be used to write a variation of the pow\_of\_2 program. What printed? Write a program to check your answer.

3id \_\_ yes o\_no\_o\_no

\_am one\_i\_aren"t me\_to-2 0~Lgo  xYshouldI

star~(i t int

i 0, power = 2048; ile «power *1=* 2) > O)

5 Design a standard form of introductory comment that will give a reader informa tion about who wrote the program and why. 

6 Take a symbol such as + and show the different ways it can be used in a program.

7 ANSI C does not provide for the nesting of comments, although many compilers provide an option for this. Try the following line on your compiler and see what happens:

/\* This is an attempt 1\* to nest \*1 a comment. \*1 

8 The ANSI C committee is conSidering adding the c++ comment style to the C lan guage, and some compilers already accept this. If you put a line such as

II Will this c++ style comment work in C?

in one of your working C programs, what happens on your system? Even if your C compiler does accept this comment style, you should be aware that many compilers do not. If you want your code to be portable, do not use *C++* style comments.

9 Write an interactive program that converts pounds and ounces to kilograms and grams. Use symbolic constants that are defined at the top of the file outside of mainO.

10 This exercise illustrates one place where white space around operators is impor tant. The expression a+++b can be interpreted as either

printf("%-6d", power);

program that you wrote in exercise 12 contains a whi 1 e loop. Write another ~."""T"'ClYYl that has the same effect, but use a for loop instead.

Study the following code and write down what you think gets printed. Then write a test program to check your answers.

i nt a, b = 0, c = O;

a = ++b + ++c;

printf("%d %d %d\n" , a, b, c) ;

a = b++ + c++;

printf("%d %d %d\n", a, b, c) ;

a = ++b + c++;

printf("%d %d %d\n" , a, b, c) ;

a = b-- + --c;

printf("%d %d %d\n" , a, b, c) ;

What is the effect in the following statement if some, or all, of the parentheses are . removed'? Explain.

x = (y 2) + (z = 3);

First complete the entries in the table that follows. After you have done this, write a program to check that the values you entered are correct. Explain the error in the last expression in the table. Is it a compile-time error or a run-time error?

100 Chapter 2 'f Lexical Elements, Operators, and the C System

Declarations and initializations

i nt a 2, b == -3, c 5, d -7, e 11;

• Expression Equivalent expression Value

! / b / c (a / b) / c 0 ,a

| '7 + c ,', d / e | 7 + ((c 'I, (-- d)) / e) |  |
| --- | --- | --- |
| !2 *i,* a % b + c + 1 |  |  |
| 39 / ++ e + 29 % c |  |
| a +== b +== C += 1 + 2 |  |  |
| 7 + ++ a % (3 + b) |  | error |

17 Consider the following code:

int a = 1, b == 2, c 3;

a += b += C += 7;

Write an equivalent statement that is fully parenthesized. What are the values of the variables a, b, and c? First write dmvn your answer. Then write a test program to check your answer.

18 A good programming style includes a good layout style, and a good layout style is crucial to the human reader, even though the compiler sees only a stream of charac ters. Consider the following program:

int main(void

){float qx,

zz,

tt;printf("gimme 3"

);scanf

("%f%f %f" ,&qx,&zz

,&tt);printf("averageis=%f",

(qx+tt+zz)/3.0);return

o

; }

Although the code is not very readable, it should compile and execute. Test it to see if that is true. Then completely reV'.Tite the program. Use white space and comments to make it more readable and well documented. *Hint:* Include a header file and choose new identifiers to replace qx, ZZ, and tt.

'f Exercises

101

the *prnJand* program in Section 2.l3, "The C System," on page 93, replac

the for loop with the following whi 1 e loop:

< n) {

you get your program running and understand its effects, rewrite the pro

changing

to ++i < n

the program will behave differently. To compensate for this, rewrite the body

whi 1 e loop so that the program behaves exactly as it did in the beginning.

integers produced by the function rand 0 all fall within the interval [0, n],

*n* is system-dependent. In ANSI C, the value for *n* is given by the symbolic

RAND\_MAX, which is defined in the standard header file *stdlib.h.* Write a

, ... ", .... ",,..... that prints out the value of RAND\_MAX on your system. *Hint:* Include the file *stdlib.h* and use the line

printf("RAND\_MAX = %d\n", RAND\_MAX);

sible, run your program on a number of different C systems. You will proba

find that RAND\_MAX has the same value on all systems. The reason for this is

the ANSI C committee suggested how the function rand 0 could be imple

d, and most compiler writers followed the committee's suggestions verbatim.

as been our experience that C systems on PCs, UNIX workstations, and even the

supercomputer in San Diego all use the same value for RAND\_MAX and that on

of these systems rand 0 produces the same output values. (See exercise 25, on 105, for further discussion.)

the *prnJand* program in Section 2.13, "The C System," on page 93, three times

print out, say, 100 randomly distributed integers. Observe that the same list of ers gets printed each time. For many applications, this is not desirable. Mod

the *prnJand* program by using 5 rand 0 to seed the random-number generator.

first few lines of your program should look like



102 Chapter 2 T Lexical Elements, Operators, and the C System

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

; nt ma; n (vo; d)

{ int i, n, seed;

seed = time(NULL);

srand(seed);

T Exercises

103

two previous exercises we used the value returned by ti me 0 to seed the ran generator. In this exercise we want to use the value returned by o to measure the time it takes to call rand O. Here is one way this can be

<stdio.h>

<stdlib.h>

<time,h>

printf(rI\n%s\n%s",

"Some randomly distributed integers

"How many do you want to see? rI); wi 11 be pri nted. " ,  On most C systems, the function call ti me (NULL) returns the number of elap

NCALLS NCOLS

NLINES i, val;

10000000 8

3

/\* number of fct calls \*/ /\* number of columns \*/ /\* number of lines \*/

seconds since 1 January 1970. (See Appendix A, "The Standard Library.") We s this value in the variable seed, and then we use the function call s rand (seed) seed the random-number generator, Repeated calls to rand 0 will generate all integers in the interval [0, RAND\_MAX], but in a mixed-up order. The value used to seed the random-number generator determines where in the mixed-up or rand 0 will start to generate numbers. If we use the value produced by ti me 0 as a seed, then the seed will be different each time we call the program, causing a differ ent set of numbers to be produced. Run this program repeatedly. You should see a different set of numbers printed each time. Do you?

begin, diff, end;

begin = time(NULL);

srand(time(NULL));

printf("\nTIMING TEST: %d calls to randO\n\n", NCALLS); for (i = 1; i <= NCALLS; ++i) {

val = randO;

if (i <= NCOLS \* NLINES) {

printf("%7d", val);

if (; % NCOLS == 0)

22 In the previous exercise, we suggested the code

}

putchar('\n');

seed = time(NULL);

else if (i == NCOLS \* NLINES + 1) printf("%7s\n\n", ", .. , ,"); }

srand(seed);

In place of these lines, most programmers would write

srand(time(NULL));

Make this change to your program, and then compile and execute it to see that it behaves the same as before.

end = time(NULL);

diff = end - begin;

printf("%s%ld\n%s%ld\n%s%ld\n%s%,10f\n\n", " end time: ", end,

begin time: rI begin,

" elapsed time: ", diff,

"time for each call:" (double) diff / NCALLS); return 0;

104 Chapter 2 'f Lexical Elements, Operators, and the C System

Here is the output on our system:

TIMING TEST: 10000000 calls to rand()

105

'f Exercises

this exercise we continue with the discussion started in exercise 20, on page 101. call to rand 0 produces a value in the interval [0, RANDJ1AX], and RAND\_MAX typi has the value 32767. Since this value is rather small, rand 0 is not useful for

11753 19570 13830

27287 432

27126

12703 12211 17405

5493 9712 4877

23634 30284 19045

23237 31480 7305

24988 32334 1114

31011 30292 28874

scientific problems. Most C systems on UNIX machines provide the program with the rand48 family of random-number generators, so called because 48-bit gets used to generate the numbers. The function d rand48 0, for exam ple, can be used to produce randomly distributed doubl es in the range [0, 1], and function 1 rand48 0 can be used to produce randomly distributed integers in

end time: 868871916

begin time: 868871900

elapsed time: 16

time for each call: 0.0000016000

The intent of this program is to print out some of the values produced by the call to rand 0 but not all of them. After all, looking at ten million numbers on the screen is not too interesting. Experiment with this program by modifying some of the #defi nes so that you can see what their effects are. For example, try making the following changes:

range [0, 231 - 1]. Typically, the function prototypes for this family of functions in *stdlib.h.* Modify the program that you wrote in exercise 20, on page 101, to 1 rand480 in place of randO and srand480 in place of srandO. You will see , on average, larger numbers are generated. Whether the numbers are better

on the application. To find out more about pseudo random-number gener- , consult the text *Numerical Recipes in* C by William Press et al. (Cambridge, LU'F>'UU~'. Cambridge University Press, 1992), pages 274-328.

value of expressions such as ++a + a++ and a += ++a are system-dependent, \C'-'AU.C'~ the side effects of the increment operator ++ can take place at different

#define #define #define

NCALLS NCOLS NLINES

1000 7

7

/\* number of fct calls \*/ /\* number of columns \*/ /\* number of lines \*/

. This is both a strength and a weakness of C. On the one hand, compilers can what is natural at the machine leveL On the other hand, because such an expres is system-dependent, the expression will have different values on different

*Caution:* If you are on a time-shared machine, then the use of values returned by ti me 0 to time things can be misleading. Between your calls to ti me 0, the machine may be servicing other requests, making your timing results inaccurate. The proper way to time C code is with the use of the clock 0 fmIction. (See Section 11.16, "How to Time C Code," on page 528.) 

24 The function randO returns values in the interval [0, RAND\_MAX]. (See exercise 20, on page 101.) If we declare the variable medi an of type double and initialize it to have the value RAND\_MAX/2 .0, then rand 0 will return a value that is sometimes larger than medi an and sometimes smaller. On average, there should be as many values that are larger as there are values that are smaller. Test this hypothesis. Write a program that calls randO, say 500 times, inside a for loop, increments the variable above\_cnt every time rand 0 returns a value larger than medi an, and increments the variable bel ow\_cnt every time rand 0 returns a value less than medi an. Each time through the for loop, print out the value of the difference of above\_cnt and bel ow\_cnt. This difference should oscillate about zero. Does it?

s. Experienced C programmers recognize expressions such as this to be y dangerous and do not use them. Experiment with your machine to see value is produced by ++a + a++ after a has been initialized to zero. Unfortu many compilers do not warn about the danger of such expressions. What *""'y, ..... ",.....* on your system?

on a UNIX system typically end in *.a,* which is mnemonic for "archive" libraries in Win 95/NT systems typically end in *.lib.* See if you can find the C libraries on your system. These libraries are not readable by humans. a UNIX system you can give a command such as

*ar t /usr/lib/libc.a*

see all the titles (names) of the objects in the library. If you do not see any math tical functions, then the mathematics library is probably in a separate file. Try command

*ar* t */usr/lib/libm.a*

106 Chapter 2... Lexical Elements, Operators, and the C System 

28 In both ANSI C and traditional C, a backslash at the end of a line in a string constant has the effect of continuing it to the next line. Here is an example of this:

"by using a backslash at the end of the line \

a string can be extended from one line to the next"

Write a program that uses this construct. Many screens have 80 characters per line. What happens if you try to print a string with more than 80 characters?

29 In ANSI C, a backslash at the end of any line is supposed to have the effect of con tinuing it to the next line. This can be expected to work in string constants and macro definitions on any C compiler, either ANSI or traditional. (See the previous exercise.) However, not all ANSI C compilers support this in a more general way. After all, except in macro definitions, this construct gets little use. Does your C compiler support this in a general way? Try the following:

#inc\

lude <stdio.h>

int mai\

nevoid)

ter 3 

he Fundamental Data **Types**

begin this chapter with a brief look at declarations, expressions, and assignment. we give a detailed explanation for each of the fundamental data types, paying par ticular attention to how C treats characters as small integers. In expressions with oper ands of different types, certain implicit conversions occur. We explain the rules for conversion and examine the cast operator, which forces explicit conversion.

{

printf("Will this work?\n"); ret\

**Declarations, Expressions, and Assignment**

urn 0;

}

30 When you invoke the compiler, the system first invokes the preprocessor. In this exercise we want to deliberately make a preprocessing error, just to see what hap pens. Try the follOwing program:

Variables and constants are the objects that a program manipulates. In C, all variables be declared before they can be used. The beginning of a program might look like

#incl <stdixx.h>

int main(void) {

*I'"* two errors on this line *'i'l*

#include <stdio.h>

int main(void)

{ int

float a, b, c;

*1\** declaration *\*1*

printf("Try me.\n"); return 0;

}

x, y = 3.3, z -7.7;

printf("Input two integers: ");

*1\** declaration with initializations *\*1*

*1\** function call *\*1*

What happens if you change #i ncl to #i ncl ude?

scanf("%d%d", &b, &c); a = b + c;

x = y + z;

*1\** function call *\*1* I'~ assi gnment 'k I *1\** assi gnment *'i'l*

108 Chapter 3 v The Fundamental Data Types 

Declarations associate a type with each variable that is declared, and this tells the com piler to set aside an appropriate amount of space in memory to hold values associated with variables. This also enables the compiler to instruct the machine to perform speci fied operations correctly. In the expression b + c, the operator + is being applied to

two variables of type i nt, which at the machine level is a different operation than + applied to variables of type float, as occurs in the expression y + z. Of course, the programmer need not be concerned that the two + operations are mechanically differ ent, but the C compiler has to recognize the difference and give the appropriate machine instructions. 

The braces { and} surround a block, which consists of declarations and statements. The declarations, if any, must occur before the statements. The body of a function defi nition is a block, but as we shall see in Section 5.10, "Scope Rules," on page 213, there are other uses for blocks.

Expressions are meaningful combinations of constants, variables, operators, and function calls. A constant, variable, or function call by itself can also be considered an expression. Some examples of expressions are

a + b

sqrt(7.333)

5.0 \* x - tan(9.0 / x)

Most expressions have a value. For example, the expression a + b has an obvious value, depending on the values of the variables a and b. If a has value 1 and b has value 2, then a + b has value 3.

The equal sign = is the basic assignment operator in C. An example of an assignment *expression* is

i = 7

The variable i is assigned the value 7, and the expression as a whole takes that value as well. When followed by a semicolon, an expression becomes a statement, or more explicitly, an expression statement. Some examples of statements are

i = 7;

printfC"The plot thickens!\n");

The following two statements are perfectly legal, but they do no useful work. Some compilers will issue warnings about such statements; others will not.

3.777;

a + b;

3.1 v Declarations, Expressions, and Assignment' 109

us consider a statement that consists of a simple assigmnent expression followed a semicolon. It will have the following form:

t, the value of the expression on the right side of the equal sign is computed. Then value is assigned to the variable on the left side of the equal sign, and this becomes value of the assignment expression as a whole. (Statements do not have a value.)

that here the value of the assignment expression as a whole is not used. That is ctly all right. The programmer is not required to use the value produced by an

Even though assignment expressions sometimes resemble mathematical equations, two notions are distinct and should not be confused. The mathematical equation

°

not become an assignment expression by typing

/\* wrong \*/

the left side of the equal sign is an expression, not a variable, and this expression not be assigned a value. Now consider the statement

current value of x is assigned the old value of x plus 1. If the old value of x is 2 the value of x after execution of the statement will be 3. Observe that as a mathe: equation

meaningless; after subtracting *x* from both sides of the equation, we obtain 0=1

*tion:* Although they look alike, the assignment operator in C and the equal sign in "'~'LL'-"llUtiCS are not comparable.

110 Chapter 3 V The Fundamental Data Types

**3.2 The Fundamental Data Types**

C provides several fundamental data types, many of which we have already seen. We need to discuss limitations on what can be stored in each type.

Fundamental data types: long form

3.3 V Characters and the Data Type char 1 1 1

fundamental types can be grouped according to functionality. The integral types those types that can be used to hold integer values; the floating types are those that be used to hold real values. They are all arithmetic types.

Fundamental types grouped by functionality

Integral types ichar signed char unsigned char ishort int long

unsigned short unsigned unsigned long

char

signed short int

signed char signed int

unsigned char signed long int

Floating types Arithmetic types

| float double long double |
| --- |
| *Integral types* + *Floating types* |

unsigned short int unsigned int unsigned long int float double long double 

~~----------------------

These are all keywords. They may not be used as names of variables. ?f course, cha stands for "character" and i nt stands for "integer," but only char and 1 nt can be as keywords. Other data types such as arrays, pointers, and structures are derived the fundamental types. They are presented in later chapters.

Usually, the keyword si gned is not used. For example, s; gned i nt is equivalent ; nt, and because shorter names are easier to type, i nt is typically used. The type char, however, is special in this regard. (See Section 3.3, "Characters and .the Da~a Type

char," on page 111.) Also, the keywords short; nt, 10l1g i nt, and unSl gned, 1 nt be, and usually are, shortened to just short, long, and unsi gned, re~pec:lvelY. keyword si gned by itself is equivalent to i nt, but it is seldom used m thIS con With all these conventions, we obtain a new list.

Fundamental data types

char signed char unsigned char

short i nt long

unsigned short unsigned unsigned long

ifloat double long double

Let us assume that the category type is defined to be anyone of the lUJllU,ClllJlCU types given in the preceding table. Using this category, we can provide the syntax of simple declaration:

declaration :: = type identifIer { , identifier }0+

These collective names are a convenience. In Chapter 6, for example, when we dis arrays, we will explain that only integral expressions are allowed as subscripts, nearuing only expressions involving integral types are allowed.

**Characters and the Data Type char**

C, variables of any integral type can be used to represent characters. In particular, char and i nt variables are used for this purpose. In some situations, an i nt may required for technical reasons. (See Section 3.9, "The Use of getcha r 0 and

charO," on page 124.) Constants such as 'a t and t + t that we think of as charac are of type; nt, not of type char. There are no constants of type char in C. This is of the few places where C++ differs from C. In C++, character constants are of type (See exercise 14, on page 142.)

In addition to representing characters, a variable of type char can be used to hold integer values. Each char is stored in memory in 1 byte. Other than being large to hold all the characters in the character set, the size of a byte is not specified C. However, on most machines a byte is composed of 8 bits and is capable, therefore, storing 28, or 256, distinct values. Only a subset of these values represents actual characters. These include the lower- and uppercase letters, digits, punctuation, special characters such as % and +. The character set also includes the white space 'hl'll'l'lf'tAl'" blank, tab, and newline.

Most machines use either ASCII or EBCDIC character codes. In the discussion that fol we will be using the ASCII code. A table for this code appears in Appendix D,

112 Chapter 3... The Fundamental Data Types

"ASCII Character Codes." For any other code, the numbers will be different, but th~ ideas are analogous. The following table illustrates the correspondence between some character and integer values on an Ascn machine:

Some character constants and their corresponding integer values

Character constants 'a' , b' 'e' ... 'z'

Corresponding values 97 98 99 ., . 112

Character constants 'A' 'B' 'e' ... 'z'

Corresponding values 65 66 67 90

Character constants '0' '1' '2 ' ... '9'

Corresponding values 48 49 50 ... 57

Character constants '&' '\*' '+'

Corresponding values 38 42 43

Observe that there is no particular relationship between the value of the character con stant representing a digit and the digit's intrinsic integer value. That is, the value of '2' is *not* 2. The property that the values for I a', 'b', I c I , and so on occur in order is important. It makes convenient the sorting of characters, words, and lines into lexico graphical order. Character arrays are needed for this kind of work. (See Section 6.10, "Strings," on page 270.)

Some nonprinting and hard-to-print characters require an escape sequence. The hori zontal tab character, for example, is 'written as \ t in character constants and in strings. Even though it is being described by the two characters \ and t, it represents a single character. The backslash character \ is called the *escape character* and is used to escape the usual meaning of the character that follows it. The following table contains some nonprinting and hard-to-print characters:

3.3 ... Characters and the Data Type char 113

Special Characters

Name of character Written in C Integer value

alert \a 7

backslash \\ 92

backspace \b 8

carriage return \r 13

double quote \" 34

formfeed \f 12

horizontal tab \t 9

newline \n 10

null character \0 0

single quote \' 39

vertical tab \v 11

question mark \7 63

alert character \a is special; it causes the bell to ring. To hear the bell, try executing program that contains the line

pri ntf("%c", '\a'); or putchar('\a');

double-quote character 11 has to be escaped if it is used as a character in a string. example is

fi' "abel! is pri nted ;, /

ly, the single-quote character ' has to be escaped if it is used in a constant char acter construct.

**1\", '\ t ');** /\* 'abc' is printed \*/

single quotes we can use either \ I! or I!

/\* 'abc' is printed \*/

double quotes we can use either \' or

printf("\'abe'"); /\* 'abc' is printed \*/

114 Chapter 3 T The Fundamental Data Types

In ANSI C, the effect of escaping an ordinary character is undefined. Some compilers will complain about this; others will not.

Another way to write a character constant is by means of a one-, two-, or three-octal digit escape sequence, as in '\007'. This is the alert character, or the audible bell. It can be written also as '\07' or '\7', but it cannot be written as '7'. ANSI C also pro vides hexadecimal escape sequences. An example is '\xla' , which is control-z.

Next, we want to understand how characters are treated as small integers, and, con versely, how small integers are treated as characters. Consider the declaration

char c:=' a';

The variable c can be printed either as a character or as an integer.

3.3 T Characters and the Data Type char 11 5

each di is a decimal digit. Note that the digits are numbered from least signifi to most significant, counting from zero. The value of the number is given by *dnxl0n* + dn\_1xl0n-1 + ... + dzxl0z + d1x101 + doxl0o

a similar fashion, strings of bits, which are comprised of the binary digits 0 and 1, be interpreted as binary numbers. The general form of a binary number, also called 2 number, is given by

printf("%c", c); printf("%d", c);

*1\** a is printed *\*1 1\** 97 is printed *\*1*

each bi is a bit, or binary digit. The value of the number is given by b*n*x2 *n* + bn-lX2n-1 + ... + b*2* x + b1x21 + box2o

Because c has an integer value, we may use it in arithmetic expressions. pri ntfC'%c%c%c", c, c + 1, c + 2); *1\** abc is printed *\*1*

Actually, in this regard there is nothing special about the type char. Any integral expression can be printed either in the format of a character or an integer.

int c; i ;

char

Now we are ready to look at how a cha r is stored in memory at the bit level. Consider declaration

c = 'a';

can think of c stored in memory in 1 byte as

for (i = 'a'; i <= 'z'; ++i) pri ntf("%c", i);

for (c = 65; c <= 90; ++c) printf("%c", c);

for (c = '0'; C <= '9'; ++c) printf("%d ", c);

*l"* abc z is pri nted ~'I /" ABC Z is printed *t'l 1\** 48 49 ... 57 is printed *\*1*

the Os and Is comprise the 8 bits in the byte. By convention, 0 represents a bit off and 1 represents a bit turned on. This string of binary digits can be consid a binary number, and its value is given by



Next, we want to look at how a char is stored in memory at the bit level, and how strings of bits are interpreted as binary numbers. Before we describe this, recall how strings of decimal digits are interpreted as decimal numbers. Consider, for example, the decimal number 20753. Its value is given by

2x104 + Oxl03 + 7xl0z + 5x101 + 3xl0o

Observe that each of the exponents corresponds to the count of a digit in the number 20753, when we count from zero starting on the right. The digit 3 with count 0 is the least significant digit, and the digit 2 with count 4 is the most significant digit. The gen eral form of a decimal number is given by

is 64 + 32 + 1, or 97, in decimal notation.

ANSI C provides the three types char, si gned char, and unsi gned char. The type char is equivalent to either si gned char or unsi gned char, depending on the com piler. Each of the three char types is stored in 1 byte, which can hold 256 distinct val ues. For a si gned char the values go from -128 to 127. For an unsi gned char the values go from 0 to 255. To determine the values that are appropriate for a plain char your system, see exercise 10, on page 141.

116 Chapter 3 T The Fundamental Data Types

**3.4 The Data Type i nt**

The data type i ntis the principal working type of the C language. This type, along

3.5 T The Integral Types short, 1ong, and unsi gned

11 7

machines with 2-byte words:

-32768 ~ -32 thousand

- 1 +32767 ~ +32 thousand

any machine. the following code is syntactically correct:

the other integral types such as char, short, and long, is designed for working the integer values that are representable on a machine.

In mathematics, the natural numbers are 0,1,2,3, ... , and these numbers, along their negatives, comprise the integers. On a machine, only a finite portion of these inte gers are representable for a given integral type.

Typically, an i nt is stored in either 2 bytes ( 16 bits) or in 4 bytes (= 32 bits).

BIG

i nt a, b a == b + c;

2000000000 1'1' 2 bi 11 i on "'1

BIG, c = BIG;

1\* out of range? \*1

are other possibilities, but this is what happens in most C systems. On older PCs, an i nt is typically stored in 2 bytes. In newer PCs, and in workstations and mainframes, an i nt is typically stored in 4 bytes. 

Because the size of an i nt varies from one C system to another, the number of tinct values that an i nt can hold is system-dependent. Suppose that we are on a puter that has 4-byte i nts. Because 4 bytes equals 32 bits, an i nt can take on 23  distinct states. Half of these states are used to represent negative integers and half used to represent nonnegative integers:

\_2 31 , \_2 31 + 1, "', 3, -2, -1, 0,1,2.3, ... ,231 1

If. on the other hand, we are using a computer that has 2-byte words. then an i nt can take on only 216 distinct states. Again, half of these states are used to represent tive integers, and half are used to represent nonnegative integers:

\_2 15 , \_2 15 + 1. ''', 3. -2, -1,0,1,2.3, ...• 215 1

Let Nmin.int represent the smallest integer that can be stored in an i nt, and let Nmax.int represent the largest integer that can be stored in an i nt. If i is a variable of type i nt, the range of values that i can take on is given by



with the end points of the range being machine-dependent. The typical situation is as follows:

On machines with 4-byte words:

Nmin int = \_2 31 = -2147483648 ~ -2 billion

Nmax-;nt = +2 31 - 1 = +2147483647 ~ +2 billion

, at run-time the variable a may be assigned an incorrect value. The logical of the expression b + c is 4 billion, which is greater than Nm,ncint. This condition called an *integer overflow.* Typically, when an integer overflow occurs, the program JlLlHU<'-O to run, but with logically incorrect results. For this reason. the programmer strive at all times to keep the values of integer expressions within the proper

addition to decimal integer constants. there are hexadecimal integer constants as 0xla and octal integer constants such as 0377. (See Section 3.12. "Hexadecimal Octal Constants." on page 134.) Many C programmers have no particular need for

eXcldeClIlaal and octal numbers. but all programmers have to know that integers that with a leading 0 are not decimal integers. For example, 11 and 011 do not have same value.

**The Integral Types short, long, and unsigned**

C, the data type i nt is considered the "natural" or "usual" type for working with inte The other integral types. such as char, short, and long, are intended for more ut:\.It"1" /.~d use. The data type short, for example, might be used in situations where is of concern. The compiler may provide less storage for a short than for an nt, although it is not required to do so. In a similar fashion. the type long might be in situations where large integer values are needed. The compiler may provide storage for along than for an i nt, although it is not required to do so. Typically, short is stored in 2 bytes and along is stored in 4 bytes. Thus, on machines with 4- words, the size of an i nt is the same as the size of along. and on machines with

1 1 8 Chapter 3 T The Fundamental Data Types

2-byte words, the size of an i nt is the same as the size of a short. If s is a variable type short, then the range of values that s can take on is given by



where typically

3.6 T The Floating Types 119

Combining long and unsigned

I Suffix Type -r Example i lu or U unsigned i37U I i

=: -32768 c::: 32 *thousand* 

+32767 c::: +32 *thousand*

If bi g is a variable of type long, then the range of values that bi g can take on is given by

| 1 or L |
| --- |
| ul or UL |

long 137L ~~I~~ i unsigned long i 37l1L



where typically

=: -2147483648 c:: -2 *billion* Nmin\_long

Nmax\_long 1 +2147483647 c:: +2 *billion*

A variable of type uns; gned is stored in the same number of bytes as an i nt. How ever, as the name implies, the integer values stored have no sign. Typically, variables of type i nt and unsi gned are stored in a machine word. If u is a variable of type uns; gned, then the range of values u can take on is given by

**The Floating Types**

C provides the three floating types: float, double, and long double. Variables this type can hold real values such as 0.001, 2.0, and 3.14159. A suffix can be ended to a floating constant to specify its type. Any unsuffixed floating constant is type doubl e. Unlike other languages, the working floating type in C is doubl e, not t.

Combining float and unsigned

o ::;; u ::;; 2wordsize 1

The typical situation is as follows: On machines with 4-byte words NmaLUnsigned = 232 - 1 = +4294967295 c::: +4 *billion*

Suffix

f or F 11 or L

| Type  float | Example  3.7F |
| --- | --- |
| long double | 3.7L |

On machines with 2-byte words 

1 = +65535 c::: +65 *thousand*

Arithmetic on unsigned variables is performed modulo 2wordsize. (See exercise 16, on page 143.)

Suffixes can be appended to an integer constant to specify its type. The type of an unsuffixed integer constant is either; nt, long, or uns; gned long. The system chooses the first of these types that can represent the value. For example, on machines with 2 -byte words, the constant 32000 is of type i nt, but 33000 is of type long.

are representable as floating constants, but they must be written with a deci point. For example, the constants 1. 0 and 2.0 are both of type double, whereas constant 3 is an i nt.

In addition to the ordinary decimal notation for floating constants, there is an expo notation, as in the example 1. 234567e5. This corresponds to the scientific 1. 234567 x 105. Recall that

1.234567 x 105 1.234567 x 10 x 10 x 10 x 10 x 10

1.234567 x 100000

*123456.7 (decimal point shifted five places)*

a similar fashion, the number 1. 234567e- 3 calls for shifting the decimal point three to the left to obtain the equivalent constant 0.001234567.

120 Chapter 3 v The Fundamental Data Types

Now we want to carefully describe the exponential notation. After we give the rules, we will show some examples. A floating constant such as 333. 77777e-22 may not contain any embedded blanks or special characters. Each part of the constant is given a name:

I Floating-point constant parts for 333. 7777e-22

3.6 v The Floating Types 121

in 4 bytes, and a daub 1 e is stored in 8 bytes. The effect of this is that a float s about 6 decimal places of accuracy, and a daub 1 e stores about 15 decimal places ccuracy. An ANSI C compiler may provide more storage for a variable of type 9 double than for one of type daub 1 e, though it is not required to do so. Many

implement along double as a daub 1 e. (See exercise 15, on page 143.) possible values that a floating type can be assigned are described in terms of called *precision* and *range.* The precision describes the number of significant

Integer 333

| Fraction  77777 | Exponent  e-22 |
| --- | --- |

r1eLllll'CU places that a floating value carries. The range describes the limits of the largest smallest positive floating values that can be represented in a variable of that type. oat on many machines has an approximate precision of 6 significant figures and approximate range of 10-38 to 10+38, This means that a positive float value is rep-

A floating constant may contain an integer part, a decimal point, a fractional part, and an exponential part. A floating constant *must* contain either a decimal point or an expo nential part or both. If a decimal point is present, either an integer part or fractional part or both *must* be present. If no decimal point is present, then there must be an inte ger part along with an exponential part.

~1>I::,llU:U in the machine in the form (only approximately true)

each di is a decimal digit; the first digit, db is positive and -38 x n x +38. The 'nr(~sent(]ltioin of a float value in a machine is actually in base 2, not base 10, but the

*floating\_constant* :: = *f\_constant*

*{f\_suffix}* opt

*. f\_part e\_part*

as presented give the correct flavor.

A doub 1 e on many machines has an approximate precision of 15 significant figures

*f\_part f\_constant* : : = 1

*Lpart*

*Lpart*

approximate range of 10-308 to 10+308. This means that a positive double value is

1 1 1

*Lpart*

*f\_part*

*f\_part e\_part e\_part*

in the machine in the form (only approximately true) ... dIS X 10n

1 *Lpart*

*Lpart* :: = *integer\_part* :: = *{digit}* 1+

*f\_part* :: = *fractionaL part* :: = *{digit} 1+ e\_part* :: = *exponentiaLpart* :: {e I E} 1 {+ 1-} *{\_suffix* :: = *floating\_suffix* :: f 1 F 1 1 I L

Some examples of floating constants are

opt *{digith+*

each di is a decimal digit, the first digit, di , is positive; and -308 ::; n ~ +308. ",n~~£'~e X is a variable of type daub 1 e. Then the statement

x = 123.45123451234512345; /\* 20 significant digits \*/ result in x being assigned a value that is stored in the form (only approximately

3.14159

314.15ge-2F

/\* of type float \*/

1. /\* equivalent to 0.0 \*/

0.123451234512345 x 10+3 *(15 significant digits)*

0e0

but not

3.14,159

314159

.e0

-3.14159

/\* equivalent to 1.0, but harder to read \*/

/\* comma not allowed \*/

/\* decimal point or exponent part needed \*/ /\* integer or fractional part needed \*/ /\* this is floating constant expression \*/

The main points you must be aware of are (l) not all real numbers are representable, (2) floating arithmetic operations, unlike the integer arithmetic operations, need not be exact. For sman computations this is usually of no concern. For very large com putations, such as numerically solving a large system of ordinary differential equations, a good understanding of rounding effects, scaling, and so on may be necessary. This is the domain of numerical analysis.

Typically, a C compiler will provide more storage for a variable of type dou b 1 ethan for one of type fl aa t, although it is not required to do so. On most machines, a fl oa t

122 Chapter 3 "If The Fundamental Data Types

**3.7 The Use of typedef**

The C language provides the typedef mechanism, which allows the programmer explicitly associate a type with an identifier. Some examples are

3.8 "If The si zeof Operator 123

an integer that represents the number of bytes needed to store the object in . An object can be a type such as i nt or float, or it can be an expression such + b, or it can be an array or structure type. The following program uses this oper On a given machine it provides precise information about the storage require for the fundamental types.

Compute the size of some fundamental types. \*/

typedef typedef typedef

char

i nt

unsigned long

uppercase;

INCHES, FEET;

size\_t; /\* found in stddef.h \*

In each of these type definitions, the named identifiers can be used later to declare abIes or functions in the same way ordinary types can be used. Thus,

printf("The pri ntf("

pri ntf("

printf("

size of some fundamental types is computed.\n\n"); char:%3u byte \n", sizeof(char));  short:%3u by tes\n" , sizeof(short));

INCHES u; length, width;

int:%3u bytes\n", sizeof(int)); pri ntfC"

uppercase

declares the variable u to be of type uppercase, which is synonymous "vith the

printf(" pri ntf(" printf("

long:%3u by tes\n" , sizeof(long)); unsigned:%3u by tes\n" , sizeof(unsigned)); float:%3u by tes\n" , s;zeof(float));

char, and it declares the variables 1 ength and wi dth to be of type INCHES, which synonymous with the type; nt.

What is gained by allowing the programmer to create a new nomenclature for

printf("long return 0;

double:%3u by tes\n" , sizeof(double)); double:%3u bytes\n", sizeof(long double));

existing type? One gain is in abbreviating long declarations. Another is having names that reflect the intended use. Furthermore, if there are system-sensitive U'-,->u>-< tions, such as an i nt that is 4 bytes on one system and 2 bytes on another, and differences are critical to the program, then the use of typedef may make the of the software easier. In later chapters, after we introduce enumeration types structure types, we will see that the typedef facility gets used routinely.

**3.8 The s i zeof Operator**

C provides the unary operator si zeof to find the number of bytes needed to store object. It has the same precedence and associativity as all the other unary operators. expression of the form

si zeof(object)

use the C language is flexible in its storage requirements for the fundamental , the situation can vary from one machine to another. However, it is guaranteed

sizeof(char) = 1

.zeof(c~ar) <= si~eof(shor~) ~ s;zeof(int) ~ sizeof(long) slzeof(slgned) = slzeof(unslgned) = s;zeof(int)  sizeof(float) ~ sizeof(double) ~ sizeof(long double)

the signed and unsigned versions of each of the integral types are guaranteed to the same size.

that we wrote si zeof( ... ) as if it were a function. It is not, however-it is an . If si zeof is being applied to a type, then parentheses are required; otherwise are optional. The type returned by the operator is typically unsigned.

-----

124 Chapter 3 T The Fundamental Data Types

**3.9 The Use of getchar() and putchar()**

In this section, we illustrate the use of getchar 0 and putchar O. These are macros defined in *stdio.h* that are used to read characters from the keyboard and to print char acters on the screen, respectively. Although there are technical differences, a macro used as a function is used. (See Section 8.6, "An Example: Macros vvith Arguments," on page 377.) These macros, as well as others, are often used when manipulating character data.

In memory, a char is stored in 1 byte, and an i nt is stored typically in either 2 or 4 bytes. Because of this, an i nt can hold all the values that can be stored in a char, more. We can think of a char as a small integer type, and, conversely, we can think an i nt as a large character type. This is a fundamental idea, and, unfortunately, a cult one for beginning C programmers.

Our next program is called *double\_out.* It reads characters one after another from standard input file, which is normally connected to the keyboard, and writes each acter twice to the standard output file, which is normally connected to the screen.

In double\_out.c

#include <stdio.h>

int main(void)

3.9 T The Use of getcharO and putcharO 125

EOF (-1)

EOF is mnemonic for "end-of-file." What is actually used to signal an end mark is system-dependent. Although the i nt value -1 is often used different can have different values. By including the file *stdio.h* and using th~ symbolic EOF, we have made the program portable. This means that the source file can d to a different system and run with no changes. The header file *stdio.h* also the macro definitions of get char 0 and putchar O.

c;

variable c has. be~n declared as an i nt rather than a char. Whatever is used to sig end of a fIle, It cannot be a value that represents a character. Because c is an it can hold all possible character values as well as the special value EOF.

= getchar()) != EOF) {

= getchar()) != EOF

LVHHJvsed of two parts. The subexpression c = getcharO gets a value from the and assigns it to the variable c, and the value of the subexpression takes on value as well. The symbols! = represent the "not equal" operator. As long as the . of the subexpression c ;: getchar 0 is not equal to EOF, the body of the whi 1 e

{

}

•

i nt c' ,

while ((c getchar())!= EOF) { putchar(c);

putchar(c);

} return 0;

IS executed. The parentheses around the subexpression c = getcharO are neces- . Suppose we had left out the parentheses and had typed

c = getchar() != EOF

of operator precedence this is equivalent to

EOF)

is syntactically correct, but not what we want.

Dissection of the *double\_out* Program • #include <stdio.h>

One line of this header file is

value of c is written to the standard output stream in the format of a character. •

126 Chapter 3 T The Fundamental Data Types 

Characters have an underlying integer-valued representation that on most C systems is the numeric value of their ASCII representation. For example, the character constant 'a' has the value 97. The values of both the lower- and uppercase letters occur in order. Because of this, the expression 'a I + 1 has the value I b' , the expression

, b' + 1 has the value I c I , and so on. Also, because there are 26 letters in the alpha bet, the expression' z' - 'a' has the value 25. Consider the expression' A I - I a' . It has a value that is the same as 'B' - 'b', which is the same as 'C I - Ie', and so on. Because of this, if the variable c has the value of a lowercase letter, then the expression c + I A' I a' has the value of the corresponding uppercase letter. These ideas are incorporated into the next program, which capitalizes all lowercase letters.

In file capitalize.c

#include <stdio.h>

int main(void)

3.10 T Mathematical Functions 127

**Mathematical Functions**

are no built-in mathematical functions in C. Functions such as

powO expO logO sinO cosO tanO

available in the mathematics library, which is conceptually part of the standard . All of these functions, except the power function powO, take a single argument type double and return a value of type double. The power function takes two argu of type double and returns a value of type double. Our next program illustrates use of s~ rt 0 and powO. It asks the user to input a value for x and then prints it along wIth the square root of x and the value of x raised to the x power.

power\_square.c

{

i nt c;

while ((c = getchar()) 1= EOF) if (c >= 'a' && c <= 'Zl) putchar(c + 'AI - la');

else putchar(c);

return 0;

}

<math.h>

<stdio.h>

int main(void)

{ double x;

pri ntf("\n%s" , liThe following will

Because of operator precedence, the expressions

C >= la' && c <= I Z ' and (c >= 'a')'&1 (c <= IZI)

are equivalent. The symbols <= represent the operator "less than or equal." The subex pression c >= I a I tests to see if the value c is greater than or equal to the value of I a' .

**H\n"**

" the square root " x raised to the "\n");

while (1) {

be computed:\n"

of x\n"

power x\n"

The subexpression c <= I Z I tests to see if the value of c is less than or equal to the value 'z!. The symbols && represent the operator "logical and." If both subexpressions

are true, then the expression

C >= la' && c <= 'Zl

is true; otherwise it is false. Thus, the expression is true if and only if c is a lowercase letter. If the expression is true, then the statement

putchar(c + lA' - la');

}

is executed, causing the corresponding uppercase letter to be printed.

printf("Input x: ");

if (scanf("%l fll, &x) ! = 1)

break;

if (x >= 0.0)

printf("\n%1Ss%22.1Se\n%1Ss%22.1Se\n%15s%22.15e\n\n" **I1 X H; X, '** "sqrt(x) = tI, sqrt(x),

"pow(x, x) = tI, pow(x, x));

else {

printf("\nSorry, your number must be nonnegative.\n"); break;

}

128 Chapter 3., The Fundamental Data Types

pri ntfC\nSye! \n\nlt);

return 0;

}

If we execute the program and enter 2 wh~n prompted, here is what appears on the screen:

The following will be computed:

the square root of x

x raised to the power x

Input x: 2

3.10 ., Mathematical Functions 129

scanf(lt%l fll, &x)

format %1 f is used in the control string because x is a dou b 1 e. A common error is use %f instead of %1 f. Notice that we typed 2 when we illustrated the use of this pro am. Equivalently, we could have typed 2.0 or 2e0 or 0. 2el. The function call

("%1 fit, &x) would have converted each of these to the same doubl e. In C code, 2 and 2 . 0 are different. The first is of type i nt, and the second is of type b 1 e. The input stream that is read by scanf 0 is *not* source code, so the rules for code do not apply. When scanf 0 reads in a doub 1 e, the number 2 is ,just as od as the number 2.0. (See Section 11.2, "The Input Function scanfO," on page

if (scanf(lt%lflt, &x) != 1)

x

sqrt(x)

pow(x, x)

Input x:

• •

2.000000000000000e+00 1.414213562373095e+00 4.000000000000000e+00



break;

function scanfO returns the number of successful conversions. To exit the whi 1 e we can type "quit" or anything else that scanfO cannot convert to a double. If value returned by scanfO is not 1, the break statement gets executed. This causes control to exit the while statement. We discuss the break statement more in Chapter 4, "Flow of Control."

**Dissection of the** *sqrLpow* **Program** 

• #include <math.h>

#include <stdio.h>

These header files contain function prototypes. In particular, m~t~.h contains the totypes for the functions in the mathematics library. Although It IS not , to do so, as an alternative to including *math.h,* we can supply our own functIOn

types:

• double sqrt(double) , pow(double, double);

This declaration should be placed in the file just above main.O. (S,ome compilers complain if the function prototype is placed in the body of mal n 0 Itself.)

• while (1) {

Because any nonzero value is considered to be true, the expression 1 creates an whi 1 e loop. We will use a break statement to exit the loop.

if (x >= 0.0)

use the square root function is defined only for nonnegative numbers, a test is to ensure that the value of x is nonnegative. A call such as sq rt (-1.0) can cause time error. (See exercise 20, on page 144.)

printf("\n%15s%22.15e\n%15s%22.15e\n%15s%22.15e\n\n",

"x := ", x,

"sqrt(x) = ", sqrt(x) , "pow(x, x) = ", pow(x, x));

that we are printing doubl e values in the format %22 .15e. This results in 1 place the left of the decimal point and 15 places to the right, 16 significant places in all. On machine, only n places are valid, where n is between 15 and 16. (The uncertainty <COim(~S about because of the translation from binary to decimal.) You can ask for lots of ';~"l~","HU(U places to be printed, but you should not believe all that you read. •

130 Chapter 3 T The Fundamental Data Types 

**The Use of abs 0 and fabs 0**

In many languages, the function abs 0 returns the absolute value of its real In this respect, C is different. In C, the function abs 0 takes an argument of type in and returns its absolute value as an i nt. Its function prototype is in stdlib.h. For matieal code, the C programmer should use fabs 0, which takes an argument of double and returns its absolute value as a double. (See exercise 25, on page 145.) function prototype is in mathh The name fabs stands for floating absolute value.

**UNIX and the Mathematics Library**

In ANSI C, the mathematics library is conceptually part of the standard library. means that you should not have to do anything special to get access to functions. However, on older UNIX systems this is often not the case. Suppose you a program in a file, say pgm.c, that uses the sq rt () function. The following LVlllllldJlll should then compile your program:

cc pgm.c

If, however, the ANSI C system is not properly connected, you will see something the following printed on the screen:

Undefined symbol: \_sqrt

This means that the linker looked through the libraries that were made available to but was unable to find a *.0* file that contained the object code for the sqrtO (See Section 11.15, "Libraries," on page 526, for further discussion of libraries.) If give the command

cc pgm.c -1m

the mathematics library will be attached, which will allow the loader to find the sary *.0* file. In -1m the letter 1 stands for "library" and the letter m stands for maties." As older versions of UNIX give way to newer versions, the necessity of the -1m option disappears.

3.11 T Conversions and Casts 131

**Conversions and Casts**

etic expression such as x + y has both a value and a type. If both x and y . type i nt, then the expression x + y also has type i nt. But if both x and y have short, then x + y is of type i nt, not short. This is because in any expression, a always gets promoted, or converted, to an i nt. In this section we want to give rules for conversions.

**Integral Promotions**

ar or short, either si gned or unsi gned, or an enumeration type can be used in expression where an i nt or unsi gned i nt may be used. (See Section 7.5, "Enumer Types," on page 345.) If all the values of the original type can be represented by nt, then the value is converted to an i nt; otherwise, it is converted to an unsi gned This is called an integral promotion. Here is an example:

char c 0= I A';

pri ntf("%c\n", c);

char variable c occurs by itself as an argument to p ri ntf O. However, because an promotion takes place, the type of the expression c is i nt, not char.

**e Usual Arithmetic Conversions**

tic conversions can occur when the operands of a binary operator are evalu Suppos~, for example, that i is an i nt and f is a float. In the expression i + f, operand 1 gets promoted to a float, and the expression i + f as a whole has type at. The rules governing this are called the usual arithmetic conversions; those rules

132 Chapter 3 V The Fundamental Data Types 

*I'he usual arithmetic conversions:*

If either operand is of type long doubl e, the other operand is converted to long Otherwise, if either operand is of type doubl e, the other operand is converted to doubl Othen-vise, if either operand is of type float, the other operand is converted to float,

Otherwise, the integral promotions are performed on both operands, and the rules are applied:

If either operand is of type unsi gned long, the other operand is converted unsi gned long.

Otherwise, if one operand has type long and the other has type unsi gned, then of two possibilities occurs:

If along can represent all the values of an unsi gned, then the operand type unsi gned is converted to long.

If along cannot represent all the values of an unsi gned, then both of operands are converted to unsi gned long.

Othervvise, if either operand has type long, the other operand is converted to lon Otherwise, if either operand has type uns i gned, the other is converted unsi gned.

Otherwise, both operands have type i nt.

This arithmetic conversion has several alternate names:

.. automatic conversion

.. implicit conversion

II coercion

III promotion

III widening

3.11 V Conversions and Casts 133

following table illustrates the idea of automatic conversion:

Declarations

char Cj short s; int i;

long 1; unsigned u; unsigned long ul;

float f; double d; long double ld;

Expression Type Expression Type

C - s / i int u 'I, 7 - i unsigned

| u 'I, 2.0 - i | double | f 1< 7 - i | float |
| --- | --- | --- | --- |
| C + 3 | int | 7 1< S ,~ ul | unsigned long |
| C + 5.0 | double | ld + C | long double |
| d + 5 | double | u ul | unsigned long |
| 2 # i / 1 | long | u 1 | *system-dependent* |

In addition to automatic conversions in mixed expressions, an automatic conversion can occur across an assignment. For example,

= i

LOILl"<:L> the value of i, which is an i nt, to be converted to a dou b 1 e and then assigned to d, and doubl e is the type of the expression as a whole. A promotion or widening such d = i will usually be well behaved, but a narrowing or demotion such as i = dean

information. Here, the fractional part of d will be discarded. If the remaining inte part does not fit into an i nt, then what happens is system-dependent.

In addition to implicit conversions, which can occur across assignments and in mixed expressions, there are explicit conversions called *casts.* If i is an i nt, then

(double) i

will cast, or convert, the value of i so that the expression has type doubl e. The variable i itself remains unchanged. Casts can be applied to expressions. Some examples are

134 Chapter 3 v The Fundamental Data Types

(long) (tAl + 1.0)

f = (float) ((int) d + 1)

d = (double) i / 3

3.12 v Hexadecimal and Octal Constants

= A X 164 + 0 x 163 + F x 162 + 3 x 161 + C x 160  10 x 164 + 0 x 163 + 15 x 162 + 3 x 161 + 12 x 160  =: 659260

135

(double) (x = 77) but not

hexadecimal numbers and their decimal equivalents are given in the following

(double) x = 77 /\* equivalent to ((double) x) = 77, error \*/

The cast operator (type) is a unary operator having the sam~ precedence and righ left associativity as other unary operators. Thus, the expressIOn

(float) ; + 3 is equivalent to ((float) i) + 3

i Hexadecimal number

2A

B3

113

Conversion to decimal

2 x 16 + A = 2 x 16 + 10 42 B x 16 + 3 11 x 16 + 3 = 179 1 x 162 + 1 x 16 + 3 = 275

because the cast operator (type) has higher precedence than +. 

**3.12 Hexadecimal and Octal Constants**

A number represented by a positional notation in base 16 is called a hexadecimal ber. There are 16 hexadecimal digits.

Hexadecimal digits and their corresponding decimal values

F

Hexadecimal digit: 0 1 9 A B C D E 15

Decimal value: 0 1 9 10 11 12 13 14

On machines that have 8-bit bytes, a byte is conveniently represented as two hexa U<O'.UU.W digits. Moreover, the representation has two Simultaneously valid interpreta First, one may consider the 8 bits in a byte as representing a number in base 2 tation. That number can be expressed uniquely as a hexadecimal number with two llt:.ACllH;,-uuUL digits. The following table lists 8-bit bytes and corresponding two-digit llt:.I\.Clll<O'-HU'UL numbers. For convenience, decimal numbers are listed, and for later refer octal numbers are also listed.

Decimal Binary Hexadecimal Octal

0 00000000 00 000

1 00000001 01 001

2 00000010 02 002

3 00000011 03 003

A positive integer written in hexadecimal notation is a string of hexadecimal digits of the form

*hn I1n-l* .•. 112 hI ho

where each *hi* is a hexadecimal digit. It has the value

*I1nx16 n* + hn.-:1 x16n- 1 + ... + *h*2 *x16*2 + 111x161 + hox16o

For example,

31 00011111 IF 037

32 00100000 20 040

188 10111100 BC 274

254 11111110 FE 376



Another interpretation of this correspondence is also useful. By definition, a nibble consists of 4 bits, so a byte is made up of 2 nibbles. Each nibble has a unique represen-

136 Chapter 3 'f' The Fundamental Data Types

tation as a single hexadecimal digit, and 2 nibbles, malting up a byte, are representable as 2 hexadecimal digits. For example,

1011 1100 corresponds to Be

C13 'f' Summary 137

*1\** Decimal, hexadecimal, octal conversions. *\*1*

#include <stdio.h>

int main(void)

{

Note that this same correspondence occurs in the table. All of this is useful when manipulating the values of variables in bit form.

The octal digits are 0, 1, 2, ... , 7. A positive integer written in octal notation is a string of digits of the form



}

printf(l!%d %x %o\n", 19, 19, 19); printf("%d %x %o\n", 0x1c, 0x1c, 0x1c); pri ntf("%d %x %o\n" , 017, 017, (17); printf("%d\n", 11 + 0x11 + (11); printf("%x\n", 2(97151);

printf("%d\n", 0xlFfFFf);

return 0;

*I'"* 19 13 23 *\*1 I"* 28 1c 34 ,~ *I  1\** 15 f 17 *\*1* r' 37 *\*1  I"* 1fffff ," *I I'"* 2097151 ,,< *I*

where each 0i is an octal digit. It has the value 

onx811 + 011-1 x811- 1 + ... + o2 x82 + olx81 + oox8o

For example,

75301 = 7 x 84 + 5 x 83 + 3 x 82 + 0 X 81 + 1 x 80

On machines that have words consisting of 24 or 48 bits, it is natural to have words consisting of "bytes" with 6 bits, each "byte" made up of 2 "nibbles" of 3 bits each. In this case a "nibble" has a unique representation as a single octal digit, and a "byte" has

a unique representation as two octal digits.

In C source code, positive integer constants prefaced with 0 represent int~gers in octal notation, and positive integer constants prefaced with 0x or 0X represent mte~ers in hexadecimal notation. Just as with decimal integer constants, octal and hexadeCimal constants may have suffixes appended to specify the type. The letters A through F and a through f are used to code hexadecimal digits.

*hexadecimaUnteger\_constant* :: h\_integer\_consta~t {.L~uft1x }opt *h\_integer\_constant* ::;: { 0x 1 0X h *{hexadeclmaLdzgzt* h+ *hexadecimaL digit* :: = 0 1 1 1 • • • 1 9 1 a I. AI. 1 f 1 F *octaUnteger\_constant* :: = *o\_integer\_constant {Lsufflx* }opt

*o\_lnteger\_constant* :: = 0 { *octaLdigit* h+

*octaL digit* :: = 0 1 1 1 • . • 1 7

*Lsuft1x* :: *integer\_suffix:* :;: { u 1 U } opt {l 1 L } opt

Let us write a program to illustrate these ideas. We will show the output of each pri ntfO statement as a comment.

machines with 2-byte i nts, the last two formats must be changed to %1 x and %1 d, . The functions p r i n t f 0 and scan f 0 use the conversion characters and x, and 0 in conversion specifications for deCimal, hexadecimal, and octal, respec- . With pri ntfO, formats of the form %x and %0 cause integers to be printed out hexadecimal and octal notation, but not prefaced with 0x or 0. The formats %#x and can be used to get the prefixes. (See Section ILl, "The Output Function pri ntfO," page 493, for further discussion.) *Caution:* When using scanfO to read in a hexa u ..... uu,'u number, do not type an 0x prefix.

**C.13 Summary**

The fundamental data types are char, short, i nt, long, unsigned versions of these, and three floating types. The type char is a I-byte integral type mostly used for representing characters.

The type i nt is designed to be the "natural" or "working" integral type. The other integral types such as short, long, and unsi gned are provided for more special ized situations.

3 Three floating types, float, double, and long double, are provided to represent real numbers. Typically, a float is stored in 4 bytes and a doub1 e in 8 bytes, The number of bytes used to store along double varies from one compiler to another. However, as compilers get updated, the trend is to store along dou b 1 e in 16 bytes. The type double, not float, is the "worldng" type.