

Bradley Jones
Peter Aitken

SEVENTH
EDITION

Covers
C11

Sams **Teach Yourself**

C

in **One Hour** a Day

SAMS

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Bradley L. Jones
Peter Aitken
Dean Miller

Sams **Teach Yourself**

C Programming

in **One Hour a Day**

Seventh Edition

SAMS

800 East 96th Street, Indianapolis, Indiana 46240

Sams Teach Yourself C Programming in One Hour a Day, Seventh Edition

Copyright © 2014 by Pearson Education

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-10: 0-7897-5199-2

ISBN-13: 978-0-7897-5199-7

Library of Congress Control Number: 2013949045

Printed in the United States of America

First Printing: October 2013

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Acquisitions Editor
Mark Taber

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Copy Editor
Apostrophe Editing
Services

Indexer
Ken Johnson

Proofreader
Anne Goebel

Technical Editor
Siddhartha Singh

Team Coordinator
Vanessa Evans

Designer
Mark Shirar

Layout
Mary Sudul

Contents at a Glance

Introduction	1
PART I: Fundamentals of C	
1 Getting Started with C	5
2 The Components of a C Program	23
3 Storing Information: Variables and Constants	37
4 The Pieces of a C Program: Statements, Expressions, and Operators	57
5 Packaging Code in Functions	91
6 Basic Program Control	119
7 Fundamentals of Reading and Writing Information	145
PART II: Putting C to Work	
8 Using Numeric Arrays	167
9 Understanding Pointers	187
10 Working with Characters and Strings	213
11 Implementing Structures, Unions, and TypeDefs	241
12 Understanding Variable Scope	279
13 Advanced Program Control	299
14 Working with the Screen, Printer, and Keyboard	325
PART III: Advanced C	
15 Pointers to Pointers and Arrays of Pointers	361
16 Pointers to Functions and Linked Lists	385
17 Using Disk Files	417
18 Manipulating Strings	455
19 Getting More from Functions	487
20 Exploring the C Function Library	505
21 Working with Memory	533
22 Advanced Compiler Use	559
PART IV: Appendixes	
A ASCII Chart	583
B C/C++ Reserved Words	589
C Common C Functions	593
D Answers	599
Index	645

Table of Contents

Introduction	1
--------------	---

PART I: Fundamentals of C

LESSON 1: Getting Started with C	5
A Brief History of the C Language	6
Why Use C?	6
Preparing to Program	8
The Program Development Cycle	8
Creating the Source Code	9
Compiling the Source Code	9
Linking to Create an Executable File	10
Completing the Development Cycle	11
Your First C Program	13
Entering and Compiling hello.c	14
Summary	18
Q&A	18
Workshop	19
Quiz	19
Exercises	20
LESSON 2: The Components of a C Program	23
A Short C Program	24
The Program's Components	25
The <code>main()</code> Function (Lines 9 Through 23)	25
The <code>#include</code> and <code>#define</code> Directives (Lines 2 and 3)	25
The Variable Definition (Line 5)	26
The Function Prototype (Line 7)	26
Program Statements (Lines 12, 13, 14, 17, 19, 20, 22, and 28)	26
The Function Definition (Lines 26 Through 29)	27
Program Comments (Lines 1, 11, 16, and 25)	27
Using Braces (Lines 10, 23, 27, and 29)	29
Running the Program	29
A Note on Accuracy	29

A Review of the Parts of a Program	30
Summary	32
Q&A	33
Workshop	33
Quiz	33
Exercises	34
LESSON 3: Storing Information: Variables and Constants	37
Understanding Your Computer's Memory	38
Storing Information with Variables	39
Variable Names	39
Numeric Variable Types	40
Variable Declarations	44
The <code>typedef</code> Keyword	45
Initializing Variables	45
Constants	47
Literal Constants	47
Symbolic Constants	48
Summary	53
Q&A	53
Workshop	54
Quiz	54
Exercises	55
LESSON 4: The Pieces of a C Program: Statements, Expressions, and Operators	57
Statements	58
The Impact of White Space on Statements	58
Creating a Null Statement	59
Working with Compound Statements	59
Understanding Expressions	60
Simple Expressions	60
Complex Expressions	60
Operators	61
The Assignment Operator	62
The Mathematical Operators	62
Operator Precedence and Parentheses	67

Order of Subexpression Evaluation	69
The Relational Operators	70
The <code>if</code> Statement	71
The <code>else</code> Clause	74
Evaluating Relational Expressions	77
The Precedence of Relational Operators	79
The Logical Operators	80
More on True/False Values	82
The Precedence of Operators	82
Compound Assignment Operators	84
The Conditional Operator	85
The Comma Operator	85
Operator Precedence Revisited	86
Summary	87
Q&A	88
Workshop	88
Quiz	89
Exercises	89
LESSON 5: Packaging Code in Functions	91
Understanding Functions	92
A Function Defined	92
A Function Illustrated	92
How a Function Works	95
Functions and Structured Programming	97
The Advantages of Structured Programming	97
Planning a Structured Program	97
The Top-Down Approach	99
Writing a Function	100
The Function Header	100
The Function Body	103
The Function Prototype	109
Passing Arguments to a Function	110
Calling Functions	110
Recursion	112
Where the Functions Belong	114

Working with Inline Functions	115
Summary	115
Q&A	116
Workshop	116
Quiz	116
Exercises	117
LESSON 6: Basic Program Control	119
Arrays: The Basics	120
Controlling Program Execution	121
The <code>for</code> Statement	121
Nesting <code>for</code> Statements	127
The <code>while</code> Statement	130
Nesting <code>while</code> Statements	134
The <code>do...while</code> Loop	136
Nested Loops	141
Summary	142
Q&A	142
Workshop	143
Quiz	143
Exercises	143
LESSON 7: Fundamentals of Reading and Writing Information	145
Displaying Information Onscreen	146
The <code>printf()</code> Function	146
The <code>printf()</code> Format Strings	146
Displaying Messages with <code>puts()</code>	155
Inputting Numeric Data with <code>scanf()</code>	156
Using Trigraph Sequences	161
Summary	162
Q&A	163
Workshop	163
Quiz	163
Exercises	164

PART II: Putting C to Work

LESSON 8: Using Numeric Arrays	167
What Is an Array?	168
Using Single-Dimensional Arrays	169
Using Multidimensional Arrays	173
Naming and Declaring Arrays	174
Initializing Arrays	178
Initializing Multidimensional Arrays	179
Summary	182
Q&A	183
Workshop	184
Quiz	184
Exercises	184
 LESSON 9: Understanding Pointers	 187
What Is a Pointer?	188
Your Computer's Memory	188
Creating a Pointer	188
Pointers and Simple Variables	189
Declaring Pointers	189
Initializing Pointers	190
Using Pointers	190
Pointers and Variable Types	193
Pointers and Arrays	194
The Array Name as a Pointer	194
Array Element Storage	195
Pointer Arithmetic	198
Pointer Cautions	202
Array Subscript Notation and Pointers	203
Passing Arrays to Functions	204
Summary	209
Q&A	210
Workshop	210
Quiz	210
Exercises	211

LESSON 10: Working with Characters and Strings	213
The <code>char</code> Data Type	214
Using Character Variables	215
Using Strings	218
Arrays of Characters	218
Initializing Character Arrays	219
Strings and Pointers	219
Strings Without Arrays	220
Allocating String Space at Compilation	220
The <code>malloc()</code> Function	221
Using the <code>malloc()</code> Function	222
Displaying Strings and Characters	226
The <code>puts()</code> Function	226
The <code>printf()</code> Function	227
Reading Strings from the Keyboard	228
Inputting Strings Using the <code>gets()</code> Function	228
Inputting Strings Using the <code>scanf()</code> Function	232
Summary	235
Q&A	235
Workshop	237
Quiz	237
Exercises	238
LESSON 11: Implementing Structures, Unions, and TypeDefs	241
Working with Simple Structures	242
Defining and Declaring Structures	242
Accessing Members of a Structure	243
Using Structures That Are More Complex	246
Including Structures Within Other Structures	246
Structures That Contain Arrays	250
Arrays of Structures	252
Initializing Structures	256
Structures and Pointers	259
Including Pointers as Structure Members	259
Creating Pointers to Structures	261
Working with Pointers and Arrays of Structures	264
Passing Structures as Arguments to Functions	267

Understanding Unions	268
Defining, Declaring, and Initializing Unions	269
Accessing Union Members	269
Creating Synonyms for Structures with <code>typedef</code>	274
Summary	275
Q&A	275
Workshop	276
Quiz	276
Exercises	277
LESSON 12: Understanding Variable Scope	279
What Is Scope?	280
A Demonstration of Scope	280
The Importance of Scope	282
Creating External Variables	282
External Variable Scope	283
When to Use External Variables	283
The <code>extern</code> Keyword	283
Creating Local Variables	285
Static Versus Automatic Variables	285
The Scope of Function Parameters	288
External Static Variables	289
Register Variables	289
Local Variables and the <code>main()</code> Function	290
Which Storage Class Should You Use?	291
Local Variables and Blocks	291
Summary	293
Q&A	293
Workshop	294
Quiz	294
Exercises	295
LESSON 13: Advanced Program Control	299
Ending Loops Early	300
The <code>break</code> Statement	300
The <code>continue</code> Statement	302

The <code>goto</code> Statement	304
Infinite Loops	307
The <code>switch</code> Statement	311
Exiting the Program	320
The <code>exit()</code> Function	320
Summary	321
Q&A	321
Workshop	322
Quiz	322
Exercises	322
LESSON 14: Working with the Screen, Printer, and Keyboard	325
Streams and C	326
What Exactly Is Program Input/Output?	326
What Is a Stream?	326
Text Versus Binary Streams	327
Predefined Streams	327
Using C's Stream Functions	328
An Example	329
Accepting Keyboard Input	329
Character Input	330
Working with Formatted Input	338
Controlling Output to the Screen	347
Character Output with <code>putchar()</code> , <code>putc()</code> , and <code>fputc()</code>	347
Using <code>puts()</code> and <code>fputs()</code> for String Output	349
Using <code>printf()</code> and <code>fprintf()</code> for Formatted Output	350
When to Use <code>fprintf()</code>	357
Using <code>stderr</code>	357
Summary	358
Q&A	359
Workshop	359
Quiz	359
Exercises	360

PART III: Advanced C

LESSON 15: Pointers to Pointers and Arrays of Pointers	361
Declaring Pointers to Pointers	362
Pointers and Multidimensional Arrays	363
Working with Arrays of Pointers	372
Strings and Pointers: A Review	372
Declaring an Array of Pointers to Type <code>char</code>	373
Pulling Things Together with an Example	375
Summary	381
Q&A	382
Workshop	382
Quiz	382
Exercises	383
LESSON 16: Pointers to Functions and Linked Lists	385
Working with Pointers to Functions	386
Declaring a Pointer to a Function	386
Initializing and Using a Pointer to a Function	387
Understanding Linked Lists	396
Basics of Linked Lists	396
Working with Linked Lists	398
A Simple Linked List Demonstration	403
Implementing a Linked List	406
Summary	415
Q&A	415
Workshop	415
Quiz	415
Exercises	416
LESSON 17: Using Disk Files	417
Relating Streams to Disk Files	418
Understanding the Types of Disk Files	418
Using Filenames	418
Opening a File	419

Writing and Reading File Data	423
Formatted File Input and Output	424
Character Input and Output	428
Direct File Input and Output	431
File Buffering: Closing and Flushing Files	435
Understanding Sequential Versus Random File Access	436
The <code>ftell()</code> and <code>rewind()</code> Functions	437
The <code>fseek()</code> Function	440
Detecting the End of a File	443
File Management Functions	445
Deleting a File	445
Renaming a File	446
Copying a File	447
Using Temporary Files	450
Summary	452
Q&A	452
Workshop	453
Quiz	453
Exercises	454
LESSON 18: Manipulating Strings	455
Determining String Length	456
Copying Strings	457
The <code>strcpy()</code> Function	457
The <code>strncpy()</code> Function	459
Concatenating Strings	461
Using the <code>strcat()</code> Function	461
Using the <code>strncat()</code> Function	462
Comparing Strings	464
Comparing Two Entire Strings	464
Comparing Partial Strings	466
Searching Strings	468
The <code>strchr()</code> Function	468
The <code>strrchr()</code> Function	470
The <code>strcspn()</code> Function	470
The <code>strspn()</code> Function	471

The <code>strpbrk()</code> Function	473
The <code>strstr()</code> Function	473
String-to-Number Conversions	474
Converting Strings to Integers	475
Converting Strings to Longs	475
Converting Strings to Long Longs	476
Converting Strings to Floating-Point Numeric Values	476
Character-Test Functions	477
ANSI Support for Uppercase and Lowercase	481
Summary	483
Q&A	483
Workshop	484
Quiz	484
Exercises	484
LESSON 19: Getting More from Functions	487
Passing Pointers to Functions	488
Type <code>void</code> Pointers	492
Using Functions That Have a Variable Number of Arguments	496
Functions That Return a Pointer	499
Summary	501
Q&A	502
Workshop	502
Quiz	502
Exercises	503
LESSON 20: Exploring the C Function Library	505
Mathematical Functions	506
Trigonometric Functions	506
Exponential and Logarithmic Functions	506
Hyperbolic Functions	507
Other Mathematical Functions	507
A Demonstration of the Math Functions	508
Dealing with Time	509
Representing Time	509
The Time Functions	510
Using the Time Functions	513

Error-Handling	516
The <code>assert()</code> Macro	516
The <code>errno.h</code> Header File	518
The <code>perror()</code> Function	519
Searching and Sorting	521
Searching with <code>bsearch()</code>	521
Sorting with <code>qsort()</code>	523
Searching and Sorting: Two Demonstrations	523
Summary	529
Q&A	529
Workshop	530
Quiz	530
Exercises	531
LESSON 21: Working with Memory	533
Type Conversions	534
Automatic Type Conversions	534
Explicit Conversions Using Typecasts	536
Allocating Memory Storage Space	538
Allocating Memory with the <code>malloc()</code> Function	539
Allocating Memory with the <code>calloc()</code> Function	540
Allocating More Memory with the <code>realloc()</code> Function	541
Releasing Memory with the <code>free()</code> Function	543
Manipulating Memory Blocks	545
Initializing Memory with the <code>memset()</code> Function	545
Copying Memory with the <code>memcpy()</code> Function	546
Moving Memory with the <code>memmove()</code> Function	546
Working with Bits	548
The Shift Operators	548
The Bitwise Logical Operators	550
The Complement Operator	552
Bit Fields in Structures	552
Summary	554
Q&A	554
Workshop	556
Quiz	556
Exercises	557

LESSON 22: Advanced Compiler Use	559
Programming with Multiple Source-Code Files	560
Advantages of Modular Programming	560
Modular Programming Techniques	560
Module Components	564
External Variables and Modular Programming	565
The C Preprocessor	567
The #define Preprocessor Directive	567
Using the #include Directive	572
Using #if, #elif, #else, and #endif	573
Using #if...#endif to Help Debug	574
Avoiding Multiple Inclusions of Header Files	575
The #undef Directive	576
Predefined Macros	576
Using Command-Line Arguments	577
Summary	580
Q&A	580
Workshop	581
Quiz	581
Exercises	582
PART 4: Appendixes	
APPENDIX A: ASCII Chart	583
APPENDIX B: C/C++ Reserved Words	589
APPENDIX C: Common C Functions	593
APPENDIX D: Answers	599
Index	645

About the Authors

Bradley L. Jones manages and directs the Developer.com Network, which includes sites such as Developer.com, CodeGuru, and DevX. He has developed systems using C, C#, C++, SQL Server, PowerBuilder, Visual Basic, HTML5, and more. His Twitter handle is @BradleyLJones.

Peter Aitken was on the faculty at Duke University Medical Center, where he cut his programming teeth developing computer programs for research. He is an experienced author in the IT field—on both applications and programming topics—with more than 70 magazine articles and 40 books to his credit. Aitken currently works as a consultant in the pharmaceutical industry.

Dean Miller is a writer and editor with more than 20 years of experience in both the publishing and licensed consumer product businesses. Over the years, he has created or helped shape a number of bestselling books and series, including *Teach Yourself in 21 Days*, *Teach Yourself in 24 Hours*, and the *Unleashed* series, all from Sams Publishing.

Acknowledgments

I'd like to thank Bradley Jones and Peter Aiken for creating an outstanding C programming tutorial that has stood strong for more than two decades, teaching hundreds of thousands how to program in the greatest language of all, C. I'd like to thank Mark Taber for the opportunity to take this book into a new format, and to Mandie Frank, San Dee Phillips, and Siddhartha Singh for taking the original text and my additions and molding it into a better product. On a personal level, thanks to my wife Fran, my kids John, Alice, and Margaret for their love and support. I'd like to dedicate my part of this edition to my two sisters, Sheryn and Rebecca, for their unparalleled strength through the adversity life throws them.

—Dean Miller

First and foremost, my thanks go to my coauthor, Brad Jones, for his hard work and dedication. I am also greatly indebted to all the people at Sams Publishing, unfortunately too many to mention by name, who helped bring this book from concept to completion.

—Peter Aitken

I'd first like to thank my wife for her continued understanding and patience as I take on such projects as the writing of books. A good book is the result of the symbiosis achieved by a number of people working together. I would like to acknowledge all the people—readers, editors, and others—who have taken the time to provide comments and feedback on previous editions of this book. By incorporating much of their feedback, I believe that we have made this the best book for easily learning to program C.

—Bradley L. Jones

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write us to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book and may not be able to reply personally to every message we receive.

When you write, please be sure to include this book's title, edition number, and authors as well as your name and contact information. We will carefully review your comments and share them with the authors and editors who worked on the book.

Email: feedback@sampublishing.com

Mail: Sams Publishing
 201 West 103rd Street
 Indianapolis, IN 46290 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

Introduction

As you can guess from the title, this book is set up so that you can teach yourself the C programming language in 22 one-hour lessons. Despite stiff competition from languages such as C++, Java, and C#, C remains the language of choice for people who are just learning programming. For reasons detailed in Lesson 1, “Getting Started with C,” you can’t go wrong in selecting C as your programming language.

You’ve made a wise decision selecting this book as your means of learning C. Although there are many books on C, this book presents C in the most logical and easy-to-learn sequence. The fact that the six previous editions have been on best-seller lists indicates that readers agree! This book is designed for you to work through the lessons in order on a daily basis. You don’t need any previous programming experience; although experience with another language, such as BASIC, might help you learn faster. Also no assumptions are made about your computer or compiler; this book concentrates on teaching the C language, regardless of whether you use a PC, a Mac, or a UNIX system.

This Book’s Special Features

This book contains some special features to aid you on your path to C enlightenment. Syntax boxes show you how to use specific C concepts. Each box provides concrete examples and a full explanation of the C command or concept. To get a feel for the style of the syntax boxes, look at the following example. (Don’t try to understand the material; you haven’t even reached Lesson 1!)

Syntax

```
#include <stdio.h>
printf( format-string[,arguments,...] );
```

`printf()` is a function that accepts a series of *arguments*, each applying to a conversion specifier in the given format string. It prints the formatted information to the standard output device, usually the display screen. When using `printf()`, you need to include the standard input/output header file, `stdio.h`.

The *format-string* is required; however, *arguments* are optional. For each argument, there must be a conversion specifier. The format string can also contain escape sequences. The following are examples of calls to `printf()` and their output.

Example 1

```
#include <stdio.h>
int main( void )
{
    printf( "This is an example of something printed!" );
}
```

Example 1 Output

This is an example of something printed!

Example 2

```
printf( "This prints a character, %c\n a number, %d\n a floating point,
%f", 'z', 123, 456.789 );
```

Example 2 Output

This prints a character, z
a number, 123
a floating point, 456.789

Another feature of this book is DO/DON'T boxes, which give you pointers on what to do and what not to do.

DO	DON'T
<p>DO read the rest of this section. It explains the Workshop sections that appear at the end of each lesson.</p>	<p>DON'T skip any of the quiz questions or exercises. If you can finish the lesson's Workshop, you're ready to move on to new material.</p>

You'll encounter Tip, Note, and Caution boxes as well. Tips provide useful shortcuts and techniques for working with C. Notes provide special details that enhance the explanations of C concepts. Cautions help you avoid potential problems.

Numerous sample programs illustrate C's features and concepts so that you can apply them in your own programs. Each program's discussion is divided into three components: the program itself, the input required and the output generated by it, and a line-by-line analysis of how the program works. These components are indicated by special icons.

Each lesson ends with a Q&A section containing answers to common questions relating to that lesson's material. There is also a Workshop at the end of each lesson. It contains quiz questions and exercises. The quiz tests your knowledge of the concepts presented in that lesson. If you want to check your answers, or if you're stumped, the answers are provided in Appendix D.

You won't learn C by just reading this book, however. If you want to be a programmer, you must write programs. Following each set of quiz questions is a set of exercises. You need to attempt each exercise. Writing C code is the best way to learn C.

The BUG BUSTER exercises are most beneficial. A bug is a program error in C. BUG BUSTER exercises are code listings that contain common problems (bugs). It's your job to locate and fix these errors. If you have trouble busting the bugs, these answers also are given in Appendix D.

As you progress through this book, some of the exercise answers tend to get long. Other exercises have a multitude of answers. As a result, later lessons don't always provide answers for all the exercises.

Conventions Used in This Book

This book uses different typefaces to help you differentiate between C code and regular English, and also to help you identify important concepts. Actual C code appears in a special `monospace` font. In the examples of a program's input and output, what the user types appears in **bold monospace**. Placeholders—terms that represent what you actually type within the code—appear in *italic monospace*. New or important terms appear in *italic*.

This page intentionally left blank

LESSON 2

The Components of a C Program

Every C program consists of several components combined in a specific way. Most of this book is devoted to explaining these various program components and how you use them. To help illustrate the overall picture, you should begin by reviewing a complete (though small) C program with all its components identified. In this lesson you learn:

- The components of a short C program
- The purpose of each program component
- How to compile and run a sample program

A Short C Program

Listing 2.1 presents the source code for `bigyear.c`. This is a simple program. All it does is accept a year of birth entered from the keyboard and calculate what year a person turns a specific age. At this stage, don't worry about understanding the details of how the program works. The point is for you to gain some familiarity with the parts of a C program so that you can better understand the listings presented later in this book.

Before looking at the sample program, you need to know what a function is because functions are central to C programming. A *function* is an independent section of program code that performs a certain task and has been assigned a name. By referencing a function's name, your program can execute the code in the function. The program also can send information, called *arguments*, to the function, and the function can return information to the main part of the program. The two types of C functions are *library functions*, which are a part of the C compiler package, and *user-defined functions*, which you, the programmer, create. You learn about both types of functions in this book.

Note that, as with all the listings in this book, the line numbers in Listing 2.1 are not part of the program. They are included only for identification purposes, so don't type them.

Input ▼

Listing 2.1 `bigyear.c` - A Program Calculates What Year a Person Turns a Specific Age

```
1:  /* Program to calculate what year someone will turn a specific age */
2:  #include <stdio.h>
3:  #define TARGET_AGE 88
4:
5:  int year1, year2;
6:
7:  int calcYear(int year1);
8:
9:  int main(void)
10: {
11:     // Ask the user for the birth year
12:     printf("What year was the subject born? ");
13:     printf("Enter as a 4-digit year (YYYY): ");
14:     scanf(" %d", &year1);
15:
16:     // Calculate the future year and display it
17:     year2 = calcYear(year1);
18:
19:     printf("Someone born in %d will be %d in %d.",
20:           year1, TARGET_AGE, year2);
21:
22:     return 0;
23: }
```

```
24:
25: /* The function to get the future year */
26: int calcYear(int year1)
27: {
28:     return(year1+TARGET_AGE);
29: }
```

Output ▼

```
What year was the subject born? 1963
Someone born in 1963 will be 88 in 2051.
```

The Program's Components

The following sections describe the various components of the preceding sample program. Line numbers are included so that you can easily identify the program parts discussed.

The `main()` Function (Lines 9 Through 23)

The only component required in every executable C program is the `main()` function. In its simplest form, the `main()` function consists of the name `main` followed by a pair of parentheses containing the word `void` ((`void`)) and a pair of braces (`{}`). You can leave the word `void` out and the program still works with most compilers. The ANSI Standard states that you should include the word `void` so that you know there is nothing sent to the `main` function.

Within the braces are statements that make up the main body of the program. Under normal circumstances, program execution starts at the first statement in `main()` and terminates at the last statement in `main()`. Per the ANSI Standard, the only statement that you need to include in this example is the `return` statement on line 22.

The `#include` and `#define` Directives (Lines 2 and 3)

The `#include` directive instructs the C compiler to add the contents of an include file into your program during compilation. An *include file* is a separate disk file that contains information that can be used by your program or the compiler. Several of these files (sometimes called *header files*) are supplied with your compiler. You rarely need to modify the information in these files; that's why they're kept separate from your source code. Include files should all have an `.h` extension (for example, `stdio.h`).

You use the `#include` directive to instruct the compiler to add a specific include file to your program during compilation. In Listing 2.1, the `#include` directive is interpreted to mean "Add the contents of the file `stdio.h`." You will almost always include one or more

include files in your C programs. Lesson 22, “Advanced Compiler Use” presents more information about include files.

The `#define` directive instructs the C compiler to replace a specific term with its assigned value throughout your program. By setting a variable at the top of your program and then using the term throughout the code, you can more easily change a term if needed by changing the single `#define` line as opposed to every place throughout the code. For example, if you wrote a payroll program that used a specific deduction for health insurance and the insurance rate changed, tweaking a variable created with `#define` named `HEALTH_INSURANCE` at the top of your program (or in a header file) would be so much easier than searching through lines and lines of code looking for every instance that had the information. Lesson 3, “Storing Information: Variables and Constants” covers the `#define` directive.

The Variable Definition (Line 5)

A *variable* is a name assigned to a location in memory used to store information. Your program uses variables to store various kinds of information during program execution. In C, a variable must be defined before it can be used. A variable definition informs the compiler of the variable’s name and the type of information the variable is to hold. In the sample program, the definition on line 4, `int year1, year2;`, defines two variables—named `year1` and `year2`—that each hold an integer value. Lesson 3 presents more information about variables and variable definitions.

The Function Prototype (Line 7)

A *function prototype* provides the C compiler with the name and arguments of the functions contained in the program. It appears before the function is used. A function prototype is distinct from a *function definition*, which contains the actual statements that make up the function. (Function definitions are discussed in more detail in “The Function Definition” section.)

Program Statements (Lines 12, 13, 14, 17, 19, 20, 22, and 28)

The real work of a C program is done by its statements. C statements display information onscreen, read keyboard input, perform mathematical operations, call functions, read disk files, and all the other operations that a program needs to perform. Most of this book is devoted to teaching you the various C statements. For now, remember that in your source code, C statements are generally written one per line and always end with a semicolon. The statements in `bigyear.c` are explained briefly in the following sections.

The `printf()` Statement

The `printf()` statement (lines 12, 13, 19, and 20) is a library function that displays information onscreen. The `printf()` statement can display a simple text message (as in lines 12 and 13) or a message mixed with the value of one or more program variables (as in lines 19-20).

The `scanf()` Statement

The `scanf()` statement (line 14) is another library function. It reads data from the keyboard and assigns that data to one or more program variables.

The program statement on line 17 calls the function named `calcYear()`. In other words, it executes the program statements contained in the function `calcYear()`. It also sends the argument `year1` to the function. After the statements in `calcYear()` are completed, `calcYear()` returns a value to the program. This value is stored in the variable named `year2`.

The `return` Statement

Lines 22 and 28 contain `return` statements. The `return` statement on line 28 is part of the function `calcYear()`. It calculates the year a person would be a specific age by adding the `#define` constant `TARGET_AGE` to the variable `year1` and returns the result to the program that called `calcYear()`. The `return` statement on line 22 returns a value of 0 to the operating system just before the program ends.

The Function Definition (Lines 26 Through 29)

When defining functions before presenting the program `bigyear.c`, two types of functions—library functions and user-defined functions—were mentioned. The `printf()` and `scanf()` statements are examples of the first category, and the function named `calcYear()`, on lines 26 through 29, is a user-defined function. As the name implies, user-defined functions are written by the programmer during program development. This function adds the value of a created constant to a year and returns the answer (a different year) to the program that called it. In Lesson 5, “Packaging Code in Functions,” you learn that the proper use of functions is an important part of good C programming practice.

Note that in a real C program, you probably wouldn't use a function for a task as simple as adding two numbers. It has been done here for demonstration purposes only.

Program Comments (Lines 1, 11, 16, and 25)

Any part of your program that starts with `/*` and ends with `*/` or any single line that begins with `//` is called a *comment*. The compiler ignores all comments, so they have absolutely no effect on how a program works. You can put anything you want

into a comment, and it won't modify the way your program operates. The first type of comment can span part of a line, an entire line, or multiple lines. Here are three examples:

```
/* A single-line comment */  
  
int a,b,c; /* A partial-line comment */  
  
/* a comment  
spanning  
multiple lines */
```

You should not use nested comments. A *nested* comment is a comment that has been put into another comment. Most compilers will not accept the following:

```
/*  
/* Nested comment */  
*/
```

Some compilers do allow nested comments. Although this feature might be tempting to use, you should avoid doing so. Because one of the benefits of C is portability, using a feature such as nested comments might limit the portability of your code. Nested comments also might lead to hard-to-find problems.

The second style of comment, the ones beginning with two consecutive forward slashes (`//`), are only for single-line comments. The two forward slashes tell the compiler to ignore everything that follows to the end of the line.

```
// This entire line is a comment  
int x; // Comment starts with slashes
```

Many beginning programmers view program comments as unnecessary and a waste of time. This is a mistake! The operation of your program might be quite clear when you write the code; however, as your programs become larger and more complex, or when you need to modify a program you wrote 6 months ago, comments are invaluable. Now is the time to develop the habit of using comments liberally to document all your programming structures and operations. You can use either style of comments you prefer. Both are used throughout the programs in the book.

DO	DON'T
<p>DO add abundant comments to your program's source code, especially near statements or functions that could be unclear to you or to someone who might have to modify it later.</p> <p>DO learn to develop a style that will be helpful. A style that's too lean or cryptic doesn't help. A style that is verbose may cause you to spend more time commenting than programming.</p>	<p>DON'T add unnecessary comments to statements that are already clear. For example, entering</p> <pre data-bbox="705 389 1057 466">/* The following prints Hello World! on the screen */ printf("Hello World!");</pre> <p>might be going a little too far, at least when you're completely comfortable with the <code>printf()</code> function and how it works.</p>

Using Braces (Lines 10, 23, 27, and 29)

You use braces `{ }` to enclose the program lines that make up every C function—including the `main()` function. A group of one or more statements enclosed within braces is called a *block*. As you see in later lessons, C has many uses for blocks.

Running the Program

Take the time to enter, compile, and run `bigyear.c`. It provides additional practice in using your editor and compiler. Recall these steps from Lesson 1, “Getting Started with C”:

1. Make your programming directory current.
2. Start your editor.
3. Enter the source code for `bigyear.c` exactly as shown in Listing 2.1, but be sure to omit the line numbers and colons.
4. Save the program file.
5. Compile and link the program by entering the appropriate command(s) for your compiler. If no error messages display, you can run the program by clicking the appropriate button in your C environment.
6. If any error messages display, return to step 2 and correct the errors.

A Note on Accuracy

A computer is fast and accurate, but it also is completely literal. It doesn't know enough to correct your simplest mistake; it takes everything you enter exactly as you entered it, not as you meant it!

This goes for your C source code as well. A simple typographical error in your program can cause the C compiler to choke, gag, and collapse. Fortunately, although the compiler isn't smart enough to correct your errors (and you'll make errors—everyone does!), it is smart enough to recognize them as errors and report them to you. (You saw in Lesson 1 how the compiler reports error messages and how you interpret them.)

A Review of the Parts of a Program

Now that all the parts of a program have been described, you can look at any program and find some similarities. Look at Listing 2.2 and see whether you can identify the different parts.

Input ▼

Listing 2.2 list_it.c – A Program to List a Code Listing with Added Line Numbers

```
1: /* list_it.c_This program displays a listing with line numbers! */
2: #include <stdio.h>
3: #include <stdlib.h>
4: #define BUFF_SIZE 256
5: void display_usage(void);
6: int line;
7:
8: int main( int argc, char *argv[] )
9: {
10:     char buffer[BUFF_SIZE];
11:     FILE *fp;
12:
13:     if( argc < 2 )
14:     {
15:         display_usage();
16:         return (1);
17:     }
18:
19:     if (( fp = fopen( argv[1], "r" )) == NULL )
20:     {
21:         fprintf( stderr, "Error opening file, %s!", argv[1] );
22:         return(1);
23:     }
24:
25:     line = (1);
26:
27:     while( fgets( buffer, BUFF_SIZE, fp ) != NULL )
28:         fprintf( stdout, "%4d:\t%s", line++, buffer );
29:
30:     fclose(fp);
31:     return 0;
32: }
```

```
33:
34: void display_usage(void)
35: {
36:     fprintf(stderr, "\nProper Usage is: " );
37:     fprintf(stderr, "\n\nlist_it filename.ext\n" );
38: }
```

Output ▼

```
C:\>list_it list_it.c
1:  /* list_it.c - This program displays a listing with line numbers! */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  #define BUFF_SIZE 256
5:  void display_usage(void);
6:  int line;
7:
8:  int main( int argc, char *argv[] )
9:  {
10:     char buffer[BUFF_SIZE];
11:     FILE *fp;
12:
13:     if( argc < 2 )
14:     {
15:         display_usage();
16:         return (1);
17:     }
18:
19:     if ( ( fp = fopen( argv[1], "r" ) ) == NULL )
20:     {
21:         fprintf( stderr, "Error opening file, %s!", argv[1] );
22:         return(1);
23:     }
24:
25:     line = 1;
26:
27:     while( fgets( buffer, BUFF_SIZE, fp ) != NULL )
28:         fprintf( stdout, "%4d:\t%s", line++, buffer );
29:
30:     fclose(fp);
31:     return (0);
32: }
33:
34: void display_usage(void)
35: {
36:     fprintf(stderr, "\nProper Usage is: " );
37:     fprintf(stderr, "\n\nlist_it filename.ext\n" );
38: }
```

Analysis ▼

The `list_it.c` program in Listing 2.2 displays C program listings that you have saved. These listings display on the screen with line numbers added.

Looking at this listing, you can summarize where the different parts are. The required `main()` function is in lines 8 through 32. Lines 2 and 3 have `#include` directives. Lines 6, 10, and 11 have variable definitions. Line 4 defines a constant `BUFF_SIZE` as 256, the stand size for buffers. The value to doing this is that if the buffer size changes, you only need to adjust this one line and all lines using this constant will automatically update. If you hardcode a number like 256, you'd have to search all your lines of code to make sure you caught all mentions.

A function prototype, `void display_usage(void)`, is in line 5. This program has many statements (lines 13, 15, 16, 19, 21, 22, 25, 27, 28, 30, 31, 36, and 37). A function definition for `display_usage()` fills lines 34 through 38. Braces enclose blocks throughout the program. Finally, only line 1 has a comment. In most programs, you should probably include more than one comment line.

`list_it.c` calls many functions. It calls only one user-defined function, `display_usage()`. The library functions that it uses are `fopen()` in line 19; `fprintf()` in lines 21, 28, 36, and 37; `fgets()` in line 27; and `fclose()` in line 30. These library functions are covered in more detail throughout this book.

Summary

This lesson was short, but it's important because it introduced you to the major components of a C program. You learned that the single required part of every C program is the `main()` function. You also learned that a program's real work is done by program statements that instruct the computer to perform your desired actions. You were also introduced to variables and variable definitions, and you learned how to use comments in your source code.

In addition to the `main()` function, a C program can use two types of subsidiary functions: library functions, supplied as part of the compiler package, and user-defined functions, created by the programmer. The next few lessons go into much more detail on many of the parts of a C program that you saw in this lesson.

Q&A

Q What effect do comments have on a program?

A Comments are for programmers. When the compiler converts the source code to object code, it throws the comments and the white space away. This means that they have no effect on the executable program. A program with a lot of comments executes just as fast as a program with few comments. Comments do make your source file bigger, but this is usually of little concern. To summarize, you should use comments and white space to make your source code as easy to understand and maintain as possible.

Q What is the difference between a statement and a block?

A A block is a group of statements enclosed in braces (`{}`). A block can be used in most places that a statement can be used.

Q How can I find out what library functions are available?

A Many compilers come with online documentation dedicated specifically to documenting the library functions. They are usually in alphabetical order. Appendix C, “Common C Functions,” lists many of the available functions. After you begin to understand more of C, it would be a good idea to read that appendix so that you don’t rewrite a library function. (There’s no use reinventing the wheel!)

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you’ve learned.

Quiz

1. What is the term for a group of one or more C statements enclosed in braces?
2. What is the one component that must be present in every C program?
3. How do you add program comments, and why are they used?
4. What is a function?
5. C offers two types of functions. What are they, and how are they different?
6. What is the `#include` directive used for?
7. Can comments be nested?
8. Can comments be longer than one line?
9. What is another name for an include file?
10. What is an include file?

Exercises

1. Write the smallest program possible.

2. Consider the following program:

```
1:  /* ex02-02.c */
2:  #include <stdio.h>
3:
4:  void display_line(void);
5:
6:  int main(void)
7:  {
8:      display_line();
9:      printf("\n Teach Yourself C In One Hour a Day!\n");
10:     display_line();
11:
12:     return 0;
13: }
14:
15: /* print asterisk line */
16: void display_line(void)
17: {
18:     int counter;
19:
20:     for( counter = 0; counter < 30; counter++ )
21:         printf("*" );
22: }
23: /* end of program */
```

- What line(s) contain statements?
- What line(s) contain variable definitions?
- What line(s) contain function prototypes?
- What line(s) contain function definitions?
- What line(s) contain comments?

3. Write an example of a comment.

4. What does the following program do? (Enter, compile, and run it.)

```
1:  /* ex02-04.c */
2:  #include <stdio.h>
3:
4:  int main(void)
5:  {
6:      int ctr;
7:
8:      for( ctr = 65; ctr < 91; ctr++ )
9:          printf("%c", ctr );
10: }
```

```
11:    printf("\n");
11:    return 0;
12: }
13: /* end of program */
```

5. What does the following program do? (Enter, compile, and run it.)

```
1:  /* ex02-05.c */
2:  #include <stdio.h>
3:  #include <string.h>
4:  int main(void)
5:  {
6:      char buffer[256];
7:
8:      printf( "Enter your name and press <Enter>:\n");
9:      fgets( buffer );
10:
11:     printf( "\nYour name has %d characters and spaces!",
12:             strlen( buffer ) );
13:
14:     return 0;
15: }
```

This page intentionally left blank

Index

Symbols

- & (ampersands)
 - address operator, 156, 362
 - AND operator, 550-551
 - initializing pointers, 190
- && (AND operator), 81
- * (asterisks)
 - indirection operator,
189-191, 259, 262, 362
 - functions returning
pointers, 499
 - passing by reference,
492
 - precedence, 386
 - multiplication operator,
48, 65
 - pointers, 190
- ** (double indirection
operator), 362, 371
- \ (backslashes)
 - escape sequences,
146-147
 - paths, 419
- \0 (null character), 218, 456
- \n (newline character), 146-
147
- { } (braces), 29
 - compound statements, 59
 - functions, 94-96, 103
 - initializing
multidimensional arrays,
179
- [] (brackets)
 - arrays, 87, 169
 - multidimensional array,
pointers, 363
 - precedence, 368
- > (greater than operator), 70
- >= (greater than or equal to
operator), 70
- >> (shift operator), 548-550
- < (less than operator), 70
- <= (less than or equal to
operator), 70
- << (shift operator), 548-550
- ^ (carets), XOR operator,
550-551
- , (comma operator), 126, 130
- ... (ellipses)
 - function prototypes, 424
 - functions with variable
numbers of arguments,
496
- (em dashes), subtraction
operator, 65
- (unary operator), 62-65
- > (indirect membership
operator), 263, 268
- = (equals symbols),
assignment operator, 45, 62
- == (equals operator), 70
- ! (exclamation point), NOT
operator, 81
- != not equal to operator, 70

/ (forward slashes)
 division operator, 65
 paths, 419
 () (parentheses), 94
 functions
 function operator, 87
 passing arguments, 110
 operator precedence, 68
 pointer declaration, 368
 % (percent symbols)
 %d specifier, 151
 %f specifier, 152
 %s specifier, printf() function, 227
 %u specifier, 151
 conversion specifiers, 147
 modulus operator, 65
 . (periods), dot operator, 243-244
 | (pipes), OR operator, 550-551
 || (OR operator), 81
 + (plus symbols), addition operator, 65
 ++ unary operator, 62, 65, 264
 # (pound symbols), stringizing operator, 569-571
 #define directive (preprocessor), 26, 574
 constants, 49
 declaring arrays, 174
 function macros, 568-572
 macros versus functions, 571-572
 substitution macros, 567-568, 572
 symbolic character constants, 215
 #define statements, 178
 #elif directive (preprocessor), 573
 #else directive (preprocessor), 573

#endif directive (preprocessor), 573
 #if directive (preprocessor), 573
 #if...#endif directive (preprocessor), 573-575
 #include directive (preprocessor), 25, 572
 #undef directive (preprocessor), 576
 ## (concatenation operator), 570
 "" (double quotation marks), format strings, 146
 : (semicolons), 130
 function headers, 101
 function prototypes, writing, 109
 ~ (tildes), complement operator, 552
 _ (underlines)
 __DATE__ macro, 576
 __FILE__ macro, 576
 __LINE__ macro, 576
 __TIME__ macro, 576

A

abs() function, 507
 accessibility variables, 280-281
 accuracy
 source code, 30
 variables. See precision variables
 acos() function, 506
 adding
 elements to linked lists
 beginning of lists, 398-399
 empty lists, 399
 end of lists, 400-401
 existing lists, 400
 middle of lists, 401-402
 links to linked lists, 398

addition operator (+), 65
 address operator (&), 156, 362
 addresses
 array elements, 195-198
 memory, 188
 RAM, 38
 addressing pointers to functions, 386
 add to list() function, linked lists, 412
 agechecker.c, 73-74
 agechecker2.c, 75-76
 allocating memory
 for strings, 221
 at compilation, 220
 malloc() function, 222-226
 storage space
 calloc() function, 540-541
 dynamic memory allocation, 538
 freeing via free() function, 543-545
 malloc() function, 539
 realloc() function, 541-543
 static memory allocation, 538
 ampersands (&)
 && (AND operator), 81
 address operator, 156, 362
 AND operator, 550-551
 initializing pointers, 190
 ANSI
 C Standard, 6, 10
 C11 Standard, 6
 compatibility, functions and, 311
 compliance, functions and, 332-334

- strings
 - comparing, 466
 - converting uppercase/lowercase characters, 481-483
 - strings, comparing, 466
- approximate range (variables), 42
- arguments, 24. *See also* variables
 - base, 523
 - buf, 431-432
 - cmp, 521
 - command-line, 577-580
 - conversion specifiers, 154
 - dest
 - memcpy() function, 546
 - memmove() function, 546
 - expression, 516
 - filename, 420
 - fp, 431, 435-437, 443
 - functions, 95
 - functions with variable numbers of arguments, 496-498
 - macros, 568
 - passing to, 110
 - writing, 100-103
 - in functions, 94
 - key, 521
 - mode, fopen() function, 420
 - msg, 519
 - num, 521
 - passing to parameters, 102
 - printf() function, 154
 - ptr, 541
 - retrieving, 497
 - scanf() function, 160, 338-339
 - size, 431-432
- src
 - memcpy() function, 546
 - memmove() function, 546
 - structures, passing as arguments to functions, 267-268
 - variables, passing arguments as, 488
 - void pointers, 492
- arithmetic expressions, typecasts and, 536-537
- arithmetic operation, pointers, 202
- arrays
 - array elements, 168
 - arrays of pointers
 - allocating memory, 372
 - char type, 372-375
 - declaring, 373
 - example of, 375-381, 386
 - initializing, 374
 - passing to functions, 374-375
 - sorting, 376
 - sorting keyboard input, 377-379
 - strings, 372
 - buf, 543
 - characters, 218
 - initializing, 219
 - pointers to, 219-220
 - declaring, 169, 174-178
 - defining, 120, 168
 - element addresses, displaying, 196
 - elements, assigning values to, 553
 - for statements, 120
 - indexes, 120
 - initializing, 178-182
 - multidimensional, 178
 - initializing, 179-182
 - memory, 173
 - passing to functions, 369-371
 - pointers, 363-368, 371
 - printing elements, 368
 - subscripts, 173
 - naming, 174-178
 - of pointers, 372, 380
 - of structures, 255
 - passing
 - by reference, 489
 - to functions, 204-209
 - pointers
 - arithmetic operations, 202
 - array element storage, 195-198
 - array names as, 194-195
 - comparisons, 201
 - decrementing, 199-201
 - differencing, 201
 - incrementing, 198-199
 - random arrays, 182
 - single-dimensional, 169-173
 - structures
 - containing arrays, 250-252
 - of arrays, 252-258, 264-266
 - subscripts, 120, 169, 203
 - two-dimensional, 173-176, 364
 - variables, 168
- ASCII
 - concatenating strings, 462
 - source code files, 9
- ASCII character set, 214-217
- ASCII code, comparing strings, 464
- asctime() function, 511, 515
- asin() function, 506
- asm, 589
- assert() function, 516-518

assignment operators, 45, 62, 84-85

assignment statements, 231
 nesting, 61
 operator precedence, 67

assignment, type conversion
 by, 535-536

asterisks (*)
 ** (double indirection operator), 362, 371

indirection operator,
 189-191, 259, 262, 362
 functions returning pointers, 499
 passing by reference, 492
 precedence, 386

multiplication operator,
 48, 65

pointers, 190

atan() function, 506

atan2() function, 506

atof() function, 476-477

atoi() function, 475

atol() function, 475

atoll() function, 476

auto, 589

auto keyword, 288

automatic type conversions
 assignments, conversion by, 535-536
 expression, type promotion, 534-535

automatic variables
 static variables versus, 285-288
 storage classes, 291

average() function, 497-498

B

B language, 6

backslashes (\)
 B language, 6

\n (newline character), 146-147

escape sequences, 146, 147
 paths, 419

base arguments, 523

binary files
 direct output, 423
 EOF, 445

binary instructions, 9

binary-mode files, 418, 431-434

binary operators, 65-67

binary streams, 327, 418

bitwise operators, 548
 bit fields, 552-554
 complement operator (~), 552
 logical operators, 550-551
 shift operators, 548-550

blocks, 29, 59-60
 if statements, 72
 local variables and, 291-293

BLOCKSIZE constant, 544

Bool, 591

boolean operators. See logical operators

braces ({}), 29
 compound statements, 59
 functions, 94-96, 103
 initializing
 multidimensional arrays, 179

brackets ([])
 > (greater than operator), 70
 >= (greater than or equal to operator), 70
 >> (shift operator), 548-550
 < (less than operator), 70
 <= (less than or equal to operator), 70
 << (shift operator), 548-550

arrays, 87, 169
 multidimensional array, pointers, 363
 precedence, 368

branching statements, goto statements, 304-306

breaking, literal string constant strings, 59

break statements, 300-302, 313-314

breaks, 589

bsearch() function, 521-529

buf arguments, 431-432

buf arrays, 543

buffered character input functions, 330

buffering files, 435-436

buffers, 435, 546

bugs, 97

by_ref() function, 491

bytes, 38

by_value() function, 491

C

C

benefits of using, 6, 7

development cycle, 12

differences between C and C++, 7

standard, most-recent changes to, 6

C extension, 9

C statements, 26
 printf(), 27
 return, 27
 scanf(), 27

C++, 7

C#, 7

calculating time difference, difftime() function, 513

calling
 functions, 92, 95-96, 110-114, 386

- printf() function, 154
- puts() function, 155
- calloc() function, 414
 - allocating memory, 540, 541
 - linked lists, 400
- carets (^), XOR operator, 550-551
- case statement, switch statements, 319-320
- cases, 589
- ceil() function, 507
- char, 589
- character input functions
 - buffered character input, 330
 - echoing, 330
 - fgetc() function, 335
 - fgets() function, 336-338
 - getc() function, 335
 - getchar() function, 330-332
 - getche() function, 334
 - getch() function, 332-334
 - gets() function, 336
 - line-input functions, 336
 - putchar() function, 333
 - text-mode files, 428-429
 - unbuffered character input, 330
 - ungetc() function, 335
- character input/output (I/O), 428
- character output
 - functions
 - fputc() function, 349
 - putc() function, 349
 - putchar() function, 347-349
 - text-mode files, 423, 430-431
- character test functions, 477-481
- characters
 - arrays of, 218
 - initializing, 219
 - pointers to, 219-220
 - char data type, 214
 - declaring variables, 215
 - defining, 213
 - displaying
 - printf() function, 227-228
 - puts() function, 226-227
 - extended ASCII, 216-217
 - initializing variables, 215
 - reading
 - gets() function, 228-232
 - scanf() function, 232-235
 - variables, 215-218
- char data type, 214
- char type
 - arrays of pointers, 372-375
 - bit fields, 552
- checkerboard array, 173
- clock() function, 513-515
- closing files, 435-436
- cmp arguments, 521
- code
 - listings. *See* listings
 - spaghetti code, 306
- comma operator (,), 85-86, 126, 130
- command-line arguments, 577-580
- comments, 27
 - nested, 28
 - writing styles, developing, 28-29
- comparing strings, 464-468
- comparison, pointers, 201
- compilers, 10
 - conditional compilation, 573
 - expressions, creating, 535
 - nested comments, 28
 - predefined macros, 576
 - preprocessor, 567
 - register keyword, 289
 - variable name length, 40
- compiling, 12, 29
 - command-line arguments, 577-580
 - errors, line numbers, 282
 - hello.c, 14-15
 - source code, 9-11
 - using graphical environments, 10
- complement operator (~), 552
- Complex, 591
- complex expressions, 60-61
- complex structures
 - arrays within structures, 250-252
 - structures within structures, 246-249
- compound assignment operators, 84-85
- compound statements, 59-60
- concatenating strings
 - strcat() function, 461-462
 - strncat() function, 462-464
- concatenation operator (##), 570
- conditional compilation, 573
- conditional debugging, #if...#endif directive (preprocessor), 574-575
- conditional operator, 85
- conditions
 - do...while loops, 137
 - do...while statements, 140
 - for statements, 122, 126
 - while statements, 130, 133
- const, 589
- const keyword, 50-52

constants

- BLOCKSIZE, 544
 - floating-point, 47
 - literal, 47-48, 60
 - pointer constants, 201
 - symbolic, 48, 60
 - #define directive, 49
 - defining, 49
 - defining with const keyword, 50-52
 - declaring arrays, 174
 - variable scope, 280
 - continue, 589
 - continue statements, 302-304
 - conversion characters, formatted output functions and, 351-352
 - conversion specifiers, 147
 - %d, 151
 - %f, 152
 - %s, printf() function, 227
 - %u, 151
 - arguments, 154
 - functions with variable numbers of arguments, 496
 - printf() function, 151-153
 - scanf() function, 339-340
 - strftime() function, 512-513
- converting
- data types, 536
 - automatic conversions, 534-535
 - explicit conversions via typecasts, 537-538
 - strings to numbers
 - double type numeric values, 476-477
 - floating point numeric values, 476
 - integers, 475
 - long longs, 476
 - longs, 475

time representations

- asctime() function, 511
- localtime() function, 510
- mk()time() function, 511
- strftime() function, 511-513

copy_file() function, 448-450

copying

- files, 447-450
- memory, memcpy() function, 546-548

strings

- strcpy() function, 457-459
- strncpy() function, 459-460

cos() function, 506

cosh() function, 507

CPU register variables, 289

CR-LF (carriage-return linefeed), 418

ctime() function, 511

ctype.h, character test functions, 477

current time, obtaining via time() function, 510

D

data storage, memory requirements, 38

data types, 44

- argument, functions with variable numbers of arguments, 496

char, 214

char type, bit fields, 552

converting, 536

- automatic conversions, 534-535
- explicit conversions via typecasts, 537-538

expressions, type

promotion, 534-535

functions

- functions with variable numbers of arguments, 496
- passing data types to, 493-495
- numeric, 41
- pointers, 193-194, 495
- size_t, 456
- variable scope, 280
- debugging via #if...#endif directive (preprocessor), 574-575
- decimal integers, 48, 156
- decimal points, constants, 48
- declaring
 - arrays, 169, 174-178
 - arrays of pointers, 373
 - character variables, 215
 - external variables, 283
 - pointers, 189-190, 262, 371
 - char type, 372
 - pointers to functions, 386, 396
 - pointers to pointers, 362-363
 - structures, 242-245
 - unions, 269
 - variables, 44, 282
- decrementing
 - counter variables, 124
 - decrement operators, 62
 - pointers, 199-201
- default, 589
- defined() operator, 574
- defining
 - functions, 27, 94-96
 - symbolic constants, 49
- degrees, trigonometric functions, 506
- delay() function, 310-311
- delaying
 - elements from linked lists, 402-403, 414

- files, 445-446
 - links from linked lists, 398
- demo() function, 105
- dereferencing void pointers, 493-495
- dest arguments, 546
- device independent programming, 326
- differencing pointers, 201
- difftime() function, 513-515
- direct access, 191
- direct file input, 431
- direct I/O, binary-mode files, 431-434
- direct output, binary files, 423
- directives (preprocessor)
 - checking, 575-576
 - #define, 26, 567, 574
 - function macros, 568-572
 - substitution macros, 572
 - #elif, 573
 - #else, 573
 - #endif, 573
 - #if, 573
 - #if...#endif, 573-575
 - #include, 25, 572
 - #undef, 576
- disk drives, file stream buffers, 435
- disk files
 - binary-mode files, 418
 - character input, 428-429
 - character I/O, 428
 - character output, 423, 430-431
 - closing files, 435-436
 - direct I/O, 431-434
 - direct output, 423
 - EOF, detecting, 443-444
 - file buffering, 435-436
 - filenames, 418-419
 - formatted input, 427-428

- formatted output, 423-426
- opening, 419-423
- random file access, 436-442
- reading/writing data, 423
- sequential file access, 436-442
- streams, 418
- text-mode files, 418
- displaying
 - strings, 227
 - printf() function, 228
 - puts() function, 226
 - times, 511
- division operator (/), 65
- do, 589
- dot operator (.), 243-244
- double, 589
- double indirection operator (**), 362, 371
- double operands, type promotion, 535
- double quotation marks (""), format strings, 146
- double type numeric values, converting strings to, 476-477
- do...while loops, 136-142, 150
 - break statements, 300-302
 - continue statements, 302-304
 - infinite loops, 307-310
 - structure, 137
- do...while statements, 140
- dynamic allocation, 220
- dynamic memory allocation, 221-226, 538

E

- echoing, 330
- editors, source code, 9
- elapsed time, calculating via difftime(), 513

- elements
 - arrays, assigning values to elements, 553
 - linked lists, 397
 - adding elements to beginning of lists, 398-399
 - adding elements to empty lists, 399
 - adding elements to end of lists, 400-401
 - adding elements to existing lists, 400
 - adding elements to middle of lists, 401-402
 - deleting elements from lists, 402-403, 414
 - pointers to, 195-198
- ellipses (...)
 - function prototypes, 424
 - functions with variable numbers of arguments, 496
- else, 589
- else clauses, 74-76
- em dashes (-)
 - (unary operator), 62-65
 - > (indirect membership operator), 263, 268
 - subtraction operator, 65
- entries/exits in functions, 108
- enum, 590
- EOF (End of File)
 - detecting, 443-444
 - fgetc() function, 430
 - putc() function, 430
- equals symbols (=)
 - == (equals operator), 70
 - assignment operator, 45, 62
- equivalence of streams, 329
- errno.h header file, 518-521
- error-handling functions
 - assert(), 516-518

- errno.h header file, 518-521
- perror(), 519-520
- errors
 - compilation, 15-17
 - fgets() function, 430
 - initializing arrays, 178
 - line numbers, 281
 - linker, 17
 - renaming files, 446
 - uninitialized pointers, dangers of, 202-203
 - variable scope, 281
- escape sequences, 146-147
 - printf() function, 147-151, 353-354
 - puts() function, 155
- examples. See listings
- exclamation points (!)
 - != not equal to operator, 70
 - NOT operator, 81
- executables, linking, 10
- executing programs
 - controlling, 121
 - do...while loops, 136-142
 - do...while statements, 140
 - for statements, 121-130
 - loops, nesting, 141-142
 - while statements, 130-136
- exit() function, 318-321
- exiting programs, 320-321
- exits/entries in functions, 108
- exp() function, 507
- expenses.c, 171-172
- exponential functions, 506-507
- expressions
 - arithmetic, typecasts and, 536-537
 - arguments, 516
 - complex, 60-61
 - conditions
 - do...while loops, 137
 - do...while statements, 140

- for statements, 122, 126
- while statements, 130, 133
- creating, 535
- increments, for statements, 122-126
- initial, 121, 126
- simple, 60
- switch statements, 311
- type promotion, 534-535
- extended ASCII characters, 216-217
- extensions
 - .C, 9
 - .o, 10
 - .obj, 10
- extern, 590
- extern keyword, 283-285, 565-566
- external static variables, 289
- external variables, 282
 - declaring, 283
 - external static, 289
 - extern keyword, 283-285
 - modular independence, 283
 - modular programming, 565-566
 - scope, 283
 - symbolic constants, 283
 - when to use, 283

F

- factorial() function, 114
- false/true values (logical operators), 81-82
- fclose() function, 435
- fcloseall() function, 436
- feof() function, 443-444
- fflush() function, 344, 436
- fgetc() function, 335, 429
- fgets() function, 336-338, 429

- field-width specifier, printf() function, 353
- file buffering, 435-436
- file extensions
 - .C, 9
 - .o, 10
 - .obj, 10
- file management, 445
- FILE structures, 420
- filenames (disk files), 418
 - arguments, 420
 - paths, 419
- files
 - closing, 435-436
 - copying, 447-450
 - defining, 326
 - deleting, 445-446
 - flushing, 435-436
 - managing
 - copying files, 447-450
 - deleting files, 445-446
 - renaming files, 446-447
 - renaming, 446-447
 - standard input/output files. See predefined streams
 - temporary, 450-452
- flags, printf() function, 353
- float, 590
- float operands, type promotion, 535
- float type pointers, 193
- float variable, 537
- floating-point constants, 47
- floating point numeric values, converting strings to, 476
- floating-point variables, 41
- floor() function, 507
- flushall() function, 436
- flushing files, 435-436
- fmod() function, 508
- fopen() function, 419-421
- for, 590

- for loops, 121, 124, 177
 - break statements, 300-302
 - character test functions, 481
 - continue statements, 302-304
 - infinite loops, 307-310
 - local variables, 287
 - pointers and arrays of structures, 264
 - strings, uppercase/lowercase character conversions, 483
- for statements
 - arrays, 120
 - executing programs, 121-126
 - nesting, 127-130
 - structure, 121
- format specifiers, literal text, 151
- format strings, 154
 - printf() function, 146
 - scanf() function, 232
- formatted input, text-mode files, 427-428
- formatted input functions, 338
 - fflush() function, 344
 - scanf() function, 338, 345-347
 - conversion specifiers, 339-340
 - extra characters, 341-343
 - precision modifiers, 340-341
 - text-mode files, 427-428
- formatted output functions
 - fprintf() function, 350-352, 357-358
 - printf() function, 350-351, 355-357
 - escape sequences, 353-354
 - field-width, 353
 - flags, 353
 - ll specifier, 352
 - l modifier, 352
 - precision, 352
 - text-mode files, 423-426
- formatting, 9
- forward slash(/)
 - division operator, 65
 - paths, 419
- fp arguments, 431, 435-437, 443
- fprintf() function, 350, 424-426
 - conversion characters, 351-352
 - stderr stream, 357-358
- fputc() function, 349, 430
- fputs() function, 349, 431
- fread() function, 432-436
- free() function, 543-545
- frexp() function, 507
- fscanf() function, 427-428
- fseek() function, 440-442, 445
- ftell() function, 437-439
- function definitions, 26
- function macros, #define directive (preprocessor), 568-572
- function prototypes, 26
- functions, 7, 24
 - abs(), 507
 - acos(), 506
 - add to list(), linked lists, 412
 - ANSI
 - compatibility, 311
 - compliance, 332-334
 - arguments, 94-95
 - functions with variable numbers of arguments, 496-498
 - passing to, 110
 - passing to parameters, 102
 - arrays of pointers, passing to functions, 374-375
 - asctime(), 511, 515
 - asin(), 506
 - assert(), 516-518
 - atan(), 506
 - atan2(), 506
 - atof(), 476-477
 - atoi(), 475
 - atol(), 475
 - atoll(), 476
 - average(), 497-498
 - body of, 94-96
 - bsearch(), 521-529
 - by_ref(), 491
 - by_value(), 491
 - calling, 92, 95, 110-111, 386
 - recursion, 112-114
 - syntax, 95-96
 - calloc(), 414
 - allocating memory, 540-541
 - linked lists, 400
 - ceil(), 507
 - character input
 - echoing, 330
 - fgetc(), 335
 - fgets(), 336-338
 - getc(), 335
 - getch(), 332-334
 - getchar(), 330-332
 - getche(), 334
 - gets(), 336
 - line-input, 336
 - putchar(), 333
 - unbuffered character input, 330
 - ungetc(), 335
 - character output
 - fputc(), 349
 - putc(), 349
 - putchar(), 347-349
 - character test, 477-481
 - clock(), 513-515

- copy_file(), 448-450
- cos(), 506
- cosh(), 507
- ctime(), 511
- data types, passing to functions, 493-495
- defining, 27, 92-96
- delay(), 310-311
- demo(), 105
- difftime(), 513-515
- entries/exits, 108
- error-handling
 - assert(), 516-518
 - errno.h header file, 518-521
 - perror(), 519-520
- exit(), 318-321
- xp(), 507
- exponential, 506-507
- factorial(), 114
- fclose(), 435
- fcloseall(), 436
- feof(), 443-444
- fflush(), 344, 436
- fgetc(), 335, 429
- fgets(), 336-338, 429
- file management, 445-446
- floor(), 507
- flushall(), 436
- fmod(), 508
- fopen(), 419-421
- formatted input
 - fflush(), 344
 - scanf(), 338-347
- formatted output
 - fprintf(), 350-352, 357-358
 - printf(), 350-357
- fprintf(), 350, 424-426
 - conversion characters, 351-352
 - stderr stream, 357-358
- fputc(), 349, 430
- fputs(), 349, 431
- fread(), 432-436
- free(), 543-545
- frexp(), 507
- fscanf(), 427, 428
- fseek(), 440-442, 445
- ftell(), 437-439
- fwrite(), 431-436
- getc(), 335, 429
- getch(), 332-334
- getchar(), 179, 182, 330-332, 429
- getche(), 334
- get_int(), 480
- get_menu_choice, 150
- gets(), 228-232, 329, 336
- half(), 110, 495
- half_of(), 111
- headers, 94-96, 100
- hyperbolic functions
 - cosh(), 507
 - sinh(), 507
 - tanh(), 507
- if statement, 111
- illustrated, 92
- independent, 282
- indirect recursion, 112
- inline, 115
- input/output functions, 329
- intcmp(), 526
- keywords, 94
- larger_of(), 108
- largest(), 206-208
- ldexp(), 507
- library functions, 10, 24
 - atof(), 476-477
 - atoi(), 475
 - atol(), 475
 - atoll(), 476
 - calloc(), 400, 540-541
 - ctime(), 511
 - error-handling, 516-521
 - fclose(), 435
 - fcloseall(), 436
 - feof(), 443-444
 - fflush(), 436
 - fgets(), 429
 - flushall(), 436
 - fopen(), 419-421
 - fprintf(), 424-426
 - fputc(), 430
 - fread(), 432-436
 - free(), 543-545
 - fscanf(), 427-428
 - fseek(), 440-442, 445
 - ftell(), 437-439
 - fwrite(), 431-436
 - getchar(), 179, 182
 - malloc(), 221-226, 379, 398-400, 414, 458, 539
 - math(), 506-509
 - print(), 227-228
 - printf(), 27, 496, 571
 - putc(), 430
 - puts(), 226-227
 - rand(), 182
 - realloc(), 541-543
 - remove(), 445-446
 - rename(), 446-447
 - rewind(), 437-439, 445
 - scanf(), 156-160, 232-235
 - search(), 521-529
 - sort(), 523-529
 - strcat(), 461-462
 - strchr(), 468-469
 - strcmp(), 464-466
 - strcpy(), 254, 457-459
 - strcspn(), 470-471
 - strlen(), 456-457
 - strncat(), 462-464
 - strncmp(), 466-468
 - strncpy(), 459-460
 - strpbrk(), 473

- strchr(), 470
- strspn(), 471-472
- strstr(), 473-474
- time, 509-515
- tmpnam(), 450-452
- ungetc(), 480
- line-input, 336
- localtime(), 510
- local variables, 285
- logarithmic functions, 506
 - log(), 507
 - log10(), 507
- macros versus, 571-572
- main(), 25, 99, 105, 114, 200, 561
 - command-line arguments, 577
 - linked lists, 412
 - local variables, 285, 290-291
- malloc(), 414
 - allocating memory, 539
 - arrays of pointers, 379
 - copying strings, 458
 - linked lists, 398-400
 - strings, 221-226
- math functions
 - abs(), 507
 - ceil(), 507
 - exponential, 506-507
 - floor(), 507
 - fmod(), 508
 - hyperbolic, 507
 - logarithmic, 506-507
 - modf(), 508
 - pow(), 508
 - sqrt(), 507
 - trigonometric, 506
 - usage examples, 508-509
- memcpy(), 546-548
- memmove(), 546-548
- memset(), 545-548
- menu(), infinite loops, 310
- mktime(), 511
- modf(), 508
- naming, 92, 100
- parameters, scope of, 288
- passing
 - arrays to, 204-209
 - by reference, 489
 - multidimensional arrays to with a pointer, 369-371
- perror(), 519-520
- placing, 114
- pointers
 - initializing, 387-396
 - void pointers, 492-496
- pointers to functions
 - calling different functions, 389-390
 - controlling sort order, 393-395
 - declaring, 386, 396
 - functions that return pointers, 499-501
 - initializing, 387-396
 - passing pointers as, 391-392, 488-492
- pow(), 508
- print(), 274
- print_report(), 150
- printf(), 10, 27, 105, 146, 201, 227-228, 355-357, 571
 - calling, 154
 - conversion characters, 351-352
 - conversion specifiers, 151-153
 - escape sequences, 147-151, 353-354
 - field-width specifier, 353
 - flags, 353
 - format strings, 146
 - functions with variable numbers of arguments, 496
 - l modifier, 352
 - ll specifier, 352
 - precision specifier, 352
- prototypes, 94-96
- putc(), 349, 430
- putchar(), 333, 347-349, 430
- puts(), 108, 155, 226-227, 349-350
- qsort(), 523-529
- rand(), 182
- realloc(), allocating memory, 541-543
- remove(), 445-446
- rename(), 446-447
- return keyword, 106
- return statements, 94-96, 106
- return types, 100
- rewind(), 437-439, 445
- scanf, 27, 159
- scanf(), 156-161, 232-235, 345-347
 - arguments, 338-339
 - conversion specifiers, 339-340
 - handling extra characters, 341-343
 - precision modifiers, 340-341
- search functions, bsearch(), 521-529
- sin(), 506
- sinh(), 507
- sleep(), 311
- sort functions, qsort(), 523-529
- sqr(), 561
- sqrt(), 507
- square(), 110
- statements, 95
- strcat(), 461-462
- strchr(), 468-469
- strcmp(), 464-466
- strcpy(), 254, 380, 457-459

strcspn(), 470-471
 streams, input/output functions, 328
 strftime(), 511-515
 string output
 fputs(), 349
 puts(), 349-350
 strlen(), 379, 456-457
 strncat(), 462-464
 strncmp(), 466-468
 strncpy(), 459-460
 strpbrk(), 473
 strrchr(), 470
 strspn(), 471-472
 strstr(), 473-474
 structured programming, 97-99
 structures, passing as arguments to functions, 267-268
 tan(), 506
 tanh(), 507
 third(), 110
 time functions, 509-510
 asctime(), 511, 515
 clock(), 513-515
 ctime(), 511
 difftime(), 513-515
 localtime(), 510
 mktime(), 511
 strftime(), 511-515
 time(), 510, 515
 usage examples, 513-515
 tmpnam(), 450-452
 trigonometric functions, 506
 ungetc(), 335, 480
 user-defined, 24, 27, 91
 usleep(), 311
 variables, 94

void pointers, 493
 void return type, 112
 writing
 arguments, 100-103
 body, 103-108
 headers, 100-103
 local variables, 103-105
 names, 100
 parameters, 100-103
 prototypes, 109
 returning values, 106-108
 return types, 100
 statements, 106
 fwrite() function, 431-436

G

garbage values, 285
 getc() function, 335, 429
 getchar() function, 179, 182, 330-332, 429
 getche() function, 334
 getch() function, 332-334
 get_int() function, 480
 get_menu_choice function, 150
 gets() function, 228, 231-232, 336
 listing, 229-230
 streams, equivalence of, 329
 global variables. See external variables
 goto statements, 304-306, 590
 grades.c, 174-176
 graphical environments, compiling, 10
 graphical IDE, command-line arguments, 579
 > (greater than operator), 70
 >= (greater than or equal to operator), 70
 >> (shift operator), 548-550

H

half() function, 110, 495
 half_of() function, 111
 HALFOF macro, 568
 hard drives, memory requirements, 38
 head pointers, 397-398
 header files, 25
 #include directive (preprocessor), 572
 modular programming, 561, 565
 preprocessor, multiple inclusions of header files, 575
 headers
 errno.h, 518-521
 functions, 94-96, 100
 of functions, 103
 writing, 101-103
 stdarg.h, functions with variable numbers of arguments, 496
 stdio.h, 150
 heaps, 543
 Hello World program, 13
 compilation errors, 15-17
 compiling hello.c, 14-15
 linker errors, 17
 hello.c source code, 14
 hexadecimal constant, 48
 hierarchical structures, structured programming, 98
 history of C, 6
 hyperbolic functions
 cosh(), 507
 sinh(), 507
 tanh(), 507

I

IDE (Integrated Development Environments), 14, 579

if, 590

if loops, perror() function and error-handling, 521

if statements, 71-77, 111

illustrated functions, 92-93

Imaginary, 591

implicit conversions, 534

include files. See header files

increment operators, 62

incrementing

- counter variables, 124
- expressions, 122
- for statements, 122, 126
- pointers, 198-200, 264-266

indenting styles, nesting loops, 141

independent functions, 282

indexes, 120

indirect access, 191

indirect membership operator (->), 263, 268

indirect recursion, 112

indirection operator (*), 189-191, 259, 262, 362

- functions returning pointers, 499
- passing by reference, 492
- precedence, 386

infinite loops, 307-310

initial expressions, 121, 126

initializing

- arrays, 178
 - character arrays, 219
 - multidimensional, 179-182
 - of pointers, 374
 - of structures, 257-258
- character variables, 215

memory, memset() function, 545-548

pointers, 190, 261-262, 400

pointers to functions, 387-396

structures, 256-258

unions, 269

variables, 45-46, 215

inline functions, 115, 590

input

- defining, 326
- device independent programming, 326
- keyboard input, 329
 - fflush() function, 344
 - fgetc() function, 335
 - fgets() function, 336-338
 - getc() function, 335
 - getchar() function, 330-332
 - getche() function, 334
 - getch() function, 332-334
 - gets() function, 336
 - line-input functions, 336
 - putchar() function, 333
 - scanf() function, 156-161, 338-347
 - ungetc() function, 335
- standard input/output files. See predefined streams
- streams
 - binary streams, 327
 - defining, 326
 - equivalence of streams, 329
 - files, 326
 - input/output functions, 328-329
 - predefined streams, 327-328
 - text streams, 327

strings

- gets() function, 228-232
- printf() function, 227-228
- puts() function, 226-227
- scanf() function, 232-235

input fields, 339

instances, defining, 245

int, 590

intcmp() function, 526

integers, converting strings to, 475

integer variables, 41, 156

I/O. See input; output

isxxxx() macros, 477-480

iteration, 114

J - K

Java, 7

key arguments, 521

keyboard

character input functions

- buffered, 330

- echoing, 330

- fgetc() function, 335

- fgets() function, 336-338

- getc() function, 335

- getchar(), 330-332

- getche() function, 334

- getch() function, 332-334

- gets() function, 336

- line-input, 336

- putchar() function, 333

- unbuffered, 330

- ungetc() function, 335

formatted input functions

- fflush() function, 344

- scanf() function, 338-347
 - reading from, scanf() function, 156-161
 - keywords, 7, 44
 - asm, 589
 - auto, 288, 589
 - Bool, 591
 - break, 589
 - case, 589
 - char, 589
 - Complex, 591
 - const, 280, 589
 - continue, 589
 - default, 589
 - do, 589
 - double, 589
 - else, 589
 - enum, 590
 - extern, 283-285, 565-566, 590
 - float, 590
 - for, 590
 - goto, 590
 - if, 590
 - Imaginary, 591
 - in functions, 94
 - inline, 590
 - int, 590
 - long, 590
 - register, 289-290, 590
 - reserved keywords list, 589-591
 - restrict, 590
 - return, 106, 590
 - short, 590
 - signed, 590
 - sizeof, 590
 - static, 286, 289, 590
 - struct, 242, 245-246, 590
 - switch, 590
 - typedef, 274-275, 590
 - union, 271-274, 590
 - unsigned, 590
 - void, 492, 590
 - volatile, 590
 - while, 591
- ## L
- l modifier, printf() function, 352
 - label statements, 306
 - larger_of() function, 108
 - largest() function, 206-208
 - ldexp() function, 507
 - length of strings, determining, 456-457
 - less than operator (<), 70
 - less than or equal to operator (<=), 70
 - library functions, 10, 24
 - atof(), 476-477
 - atoi(), 475
 - atol(), 475
 - atoll(), 476
 - calloc(), 400, 540-541
 - ctime(), 511
 - error-handling, 516-521
 - fclose(), 435
 - fcloseall(), 436
 - feof(), 443-444
 - fflush(), 436
 - fgets(), 429
 - flushall(), 436
 - fopen(), 419-421
 - fprintf(), 424-426
 - fputc(), 430
 - fread(), 432-436
 - free(), 543-545
 - fscanf(), 427-428
 - fseek(), 440-442, 445
 - ftell(), 437-439
 - fwrite(), 431-436
 - getchar(), 179, 182
 - malloc(), 221-226, 379, 398-400, 414, 458, 539
 - math(), 506-509
 - print(), 227-228
 - printf(), 27, 496, 571
 - putc(), 430
 - puts(), 226-227
 - rand(), 182
 - realloc(), 541-543
 - remove(), 445-446
 - rename(), 446-447
 - rewind(), 437-439, 445
 - scanf(), 156-160, 232-235
 - search(), 521-529
 - sort(), 523-529
 - strcat(), 461-462
 - strchr(), 468-469
 - strcmp(), 464-466
 - strcpy(), 254, 457-459
 - strcspn(), 470-471
 - strlen(), 456-457
 - strncat(), 462-464
 - strncmp(), 466-468
 - strncpy(), 459-460
 - strpbrk(), 473
 - strrchr(), 470
 - strspn(), 471-472
 - strstr(), 473-474
 - time, 509-515
 - tmpnam(), 450-452
 - ungetc(), 480
 - lifetime variables, 280
 - line-input functions, 336
 - linked lists, 396-397
 - adding links, 398
 - deleting links, 398
 - elements, 397
 - adding elements to beginning of lists, 398-399
 - adding elements to empty lists, 399
 - adding elements to end of lists, 400-401

- adding elements to existing lists, 400
- adding elements to middle of lists, 401-402
- deleting elements from lists, 402-403, 414
- example of, 403-406
- head pointers, 397
- implementing, 406-414
- links, 397
- lists of characters, 407-414
- loops, 406
- modifying links, 398
- nodes, 397
- structures, 396
- types of, 396
- linkers, 11, 17
- linking, 10-12
- Linux source code editors, 9
- list_it.c, 30-32
- LIST0403.c, 73-74
- listings
 - arithmetic expressions, 537
 - arrays
 - displaying element addresses, 196
 - expenses.c, 171-172
 - grades.c, 174-176
 - of structures, 254-255
 - arrays of pointers
 - initializing char type, 374
 - passing to function, 374-375
 - sorting keyboard input, 377-379
 - break statements, 300-301
 - command-line arguments, 578-580
 - continue statement, 303-304
 - copy_file() function, 448-450
 - copying files, 448-450
 - data types, converting, 537
 - disk files, opening, 421-422
 - EOF, detecting, 443-444
 - error-handling functions
 - assert(), 517-518
 - perror(), 520
 - external variables, extern keyword, 284
 - feof() function, 443-444
 - fgets() function, 337
 - file I/O, reading formatted data, 427-428
 - files
 - copying, 448-450
 - deleting, 445-446
 - renaming, 447
 - temporary, 451
 - fread() function, 433-436
 - fseek() function, 440-442
 - ftell() function, 438-439
 - functions
 - passing by value/reference, 490-491
 - returning pointers, 500-501
 - variable-size arguments, 497-498
 - fwrite() function, 433-436
 - getchar() function, 330-331
 - getch() function, 333-334
 - gets() function, 229-230
 - goto statement, 305-306
 - hello.c, 14-16
 - if statements, 73-74
 - if statement with else clause, 75-76
 - infinite loops, 308-310
 - linked lists
 - basic elements, 404-405
 - list of characters, 407-414
 - list_it.c, 30-32
 - local variables
 - defining, 292
 - static versus automatic, 286-287
 - logical operator precedence, 83-84
 - malloc() function, 223-224
 - math functions, 508-509
 - memory
 - allocating via calloc() function, 540-541
 - allocating via realloc() function, 542-543
 - copying, 546-548
 - free() function, 543-545
 - initializing, 546-548
 - moving, 546-548
 - modular programming, 560-561
 - modulus operator, 66
 - multidimensional arrays
 - determining size of, 366
 - passing to functions with, 369-371
 - pointer arithmetic, 367
 - random.c, 180-182
 - relationship to pointers, 365
 - multiply.c, 24-25
 - numeric nature of char variables, 215-216
 - passing arrays to functions, 205-207
 - pointers
 - arithmetic, 199-200
 - incrementing, 265-266
 - to functions, 387-395
 - usage, 191-192
 - preprocessor
 - directives, # operator in macro, 571
 - header files, 575
 - printf() function, 354-357
 - displaying numerical values, 152-153

- escape sequences, 148-149
- printing extended ASCII characters, 216-217
- putchar() function, 348-349
- puts() function, 227, 350
- relational expressions, 78
- remove() function, 445-446
- rename() function, 447
- rewind() function, 438-439
- scanf function, 234
- scanf() function, 345-347
 - conversion specifiers, 340
 - precision modifiers, 340-341
 - reading numerical values, 157-160
- search functions, 523-529
- shift operators, 549-550
- simplestruct.c, 244-245
- sort functions, 523-529
- stdin, clearing of extra characters, 342-344
- strings
 - bsearch() function, 526-529
 - comparing with strcmp() function, 465-466
 - comparing with strncmp() function, 466-468
 - concatenating via strcat() function, 461-462
 - concatenating via strncat() function, 463-464
 - converting to double type numeric values, 476-477
 - converting to floating point numeric values, 476
 - converting to integers, 475
- converting uppercase/lowercase characters, 482-483
- copying via strcpy() function, 458-459
- copying via strncpy() function, 459-460
- determining length of, 456-457
- isxxx() macros, 478-480
- qsort() functions, 526-529
- searching for first occurrence of characters, 469
- searching for first occurrence of characters in second strings, 470-471
- searching for nonmatching characters, 472
- searching for strings within strings, 473-474
- structures
 - arrays within structures, 251-252
 - passing structures as arguments to functions, 267-268
 - structures of structures, 248-249
- switch statements, 312, 317-318
 - break statements and, 313-314
 - executing menu systems, 314-317
- temporary files, 451
- time functions, 513-515
- tmpnam() function, 451
- typecasts, arithmetic expressions, 537
- unary.c, 63-64
- union members, accessing, 270-271
- unions, 272-274
 - using fprintf() function, 425-426
- variables
 - constants, 51-52
 - scope, 280-281
 - size program, 42-43
 - void pointers, 493-495
- literal constants, 47-48, 60, 174
- literal string constants
 - breaking lines, 59
 - white space, 58
- literal strings, 219
- literal text, 146
- ll specifier, printf() function, 352
- local scope, function parameters, 288
- local variables, 94, 282
 - blocks and, 291-293
 - creating, 285
 - functions, writing, 103-105
 - main function, 290-291
 - static versus automatic, 285-288
- localtime() function, 510
- logarithmic functions, 506
 - log(), 507
 - log10(), 507
- logical operators, 80, 550-552
 - precedence, 82-84
 - true/false values, 81-82
- long longs
 - converting strings to, 476
 - type promotion, 535
- longs, 590
 - converting strings to, 475
 - type promotion, 535
- loops
 - do...while, 136, 138-142, 150
 - break statements, 300-302

- continue statements, 302-304
 - infinite loops, 307-310
 - structure, 137
 - ending early, 300-301
 - for loops, 121, 124, 177
 - break statements, 300-302
 - character test functions, 481
 - continue statements, 302-304
 - infinite loops, 307-310
 - local variables, 287
 - pointers and arrays of structures, 264
 - uppercase/lowercase character conversions in strings, 483
 - if loops, perror() function and error-handling, 521
 - infinite, 307-310
 - linked lists, 406
 - nesting, 141-142
 - while loops, 132, 150
 - break statements, 300-302
 - continue statements, 302-304
 - converting strings to double type numeric values, 477
 - copying files, 450
 - detecting EOF, 444
 - infinite loops, 307-310
 - linked lists, 412
 - partial string comparisons, 468
 - random file access, 442
 - lowercase/uppercase characters, ANSI support for, 481-483
- M**
- machine language, 9
 - macros
 - functions, #define directive (preprocessor), 568-572
 - HALFOF, 568
 - isxxx(), 477-480
 - NDEBUG, 518
 - predefined macros
 - __DATE__, 576
 - __FILE__, 576
 - __LINE__, 576
 - __TIME__, 576
 - substitution macros
 - creating symbolic constants, 567-568, 572
 - #define directive (preprocessor), 567-568, 572
 - tolower(), 481-483
 - toupper(), 481-483
 - va arg(), 497
 - va end(), 497-499
 - va list, 497
 - va start(), 497-499
 - main() function, 25, 99, 105, 114, 200, 561
 - command-line arguments, 577
 - linked lists, 412
 - local variables, 285, 290-291
 - main modules, 560-561, 565
 - malloc() function, 414
 - allocating memory, 539
 - arrays of pointers, 379
 - copying strings, 458
 - linked lists, 398-400
 - listing, 223-224
 - strings, 221-226
 - managing files
 - copying files, 447-450
 - deleting files, 445-446
 - renaming files, 446-447
 - manifest constants. See symbolic constants
 - math functions
 - abs(), 507
 - ceil(), 507
 - exponential functions, 506
 - exp(), 507
 - frexp(), 507
 - ldexp(), 507
 - floor(), 507
 - fmod(), 508
 - hyperbolic functions
 - cosh(), 507
 - sinh(), 507
 - tanh(), 507
 - logarithmic functions, 506
 - log(), 507
 - log10(), 507
 - modf(), 508
 - pow(), 508
 - sqrt(), 507
 - trigonometric functions
 - acos(), 506
 - asin(), 506
 - atan(), 506
 - atan2(), 506
 - cos(), 506
 - sin(), 506
 - tan(), 506
 - usage examples, 508-509
 - mathematical operators
 - binary, 65-67
 - unary, 62-65
 - member operator (.). See dot operator (.)
 - members (structures)
 - accessing, 243-244
 - pointers as, 259-261
 - members (unions), accessing, 269-271
 - memcpy() function, 546-548

- memmove() function, 546-548
- memory, 38
 - addresses, 188
 - allocation for strings
 - at compilation, 220-221
 - malloc() function, 221-226
 - arrays, 169
 - arrays of pointers, 372
 - binary-mode files, 431
 - bitwise operators
 - bit fields, 552-554
 - complement operator (~), 552
 - logical operators, 550-551
 - shift operators, 548-550
 - buffers, 546
 - bytes, 38
 - copying, memcpy() function, 546-548
 - data storage, space requirements, 38
 - dynamic allocation, 220
 - file buffers, 435
 - freeing via free() function, 543-545
 - heaps, 543
 - initializing, memset() function, 545-548
 - linked lists, 414
 - deleting elements, 403
 - freeing memory, 412
 - memory leaks, 403
 - moving, memmove() function, 546-548
 - multidimensional arrays, 173
 - numeric variables, 40-44
 - pointers, 190-192
 - arithmetic operations, 202
 - array element storage, 195-198
 - array names as, 194-195
 - comparisons, 201
 - creating, 188-189
 - dangers of uninitialized, 202-203
 - data types, 193-194
 - declaring, 189-190
 - decrementing, 199-201
 - differencing, 201
 - incrementing, 198-199
 - initializing, 190
 - passing arrays to functions, 204-209
- RAM
 - addresses, 38, 188
 - allocating memory storage space, 539
- register variables, 289
- stacks, 488
- storage space, allocating
 - calloc() function, 540-541
 - dynamic memory allocation, 538
 - malloc() function, 539
 - realloc() function, 541-543
 - static memory allocation, 538
- type conversions
 - automatic conversions, 534-536
 - explicit conversions via typecasts, 536-538
- memset() function, 545-548
- menu() function, infinite loops, 310
- menu systems, executing with switch statements, 314-317
- menus, structured programming, 99
- messages, displaying on screen, 146
- military time, 511
- mktime() function, 511
- mode
 - arguments, fopen() function, 420
 - values, 420
- modf() function, 508
- modifying links in linked lists, 398
- modular independence, external variables, 283
- modularity, 7
- modular programming, 562-564. *See also* structured programming
 - advantages of, 560
 - external variables, 565-566
 - header files, 561, 565
 - main modules, 560-561, 565
 - secondary modules, 560-561, 564
- modules, 560-561
 - main, 565
 - secondary, 564
- modulus operator (%), 65
- moving memory, memmove() function, 546-548
- msg arguments, 519
- multidimensional arrays, 178
 - functions, passing arrays to, 369-371
 - initializing, 179-182
 - memory, 173
 - pointers, 363-365, 368, 371
 - determining size of, 366
 - pointer arithmetic, 367
 - printing elements, 368
 - subscripts, 173
- multiple indirection, 363
- multiplication operator (*), 48, 65

N

naming

- arrays, 174-178
- functions, 92, 100
- naming conventions, 40
- pointers, 188
- source files, 9
- variables, 39-40

NDEBUG macro, 518

nesting

- #include directive (preprocessor), 572
- comments, 28
- for statements, 127-130
- #include directive (preprocessor), 572
- loops, 141-142
- statements
 - assignment statements, 61
 - if, 77
 - while, 134-136

newline character (\n), 146-147, 418

nodes, linked lists, 397

not equal to operator (!=), 70

NOT operator (!), 81

null character (\0), 218, 456

null statements, 59

NULL values

- fopen() function, 420
- head pointers, 397-398
- strpbrk() function, 473
- strstr() function, 473

num arguments, 521

numerical data, 41, 157-160

numerical values, displaying with printf() function, 152-153

numeric variables, 40-44

- floating-point, 41

- integer, 41
- register keyword, 290
- void pointers, 492

O

o extension, 10

.obj extension, 10

object code, 10

object files, 10

object-oriented programming, 7

objectives, programming steps, 8

octal integers, 48

op keyword, 84

opening disk files, 419-423

operands

- modes, 63
- type promotion, 535

operating systems, memory allocation, 539

operators, 61

- address of (&), pointers, 362

AND (&), 550-551

assignment operators, 62, 84-85

binary, 65-67

bitwise operators

- bit fields, 552-554
- complement operator (~), 552

- logical operators, 550-551

shift operators, 548-550

comma (,), 85-86, 126, 130

complement (~), 552

concatenation (##), 570

conditional, 85

decrement, 62

defined(), 574

dot (.), 243-244

double indirection (**), 362

double indirection operator (**), 371

>> shift operator, 548-550

increment, 62

indirection (*), 189-191, 259, 262, 386

indirect membership (->), 263, 268

logical, 80

- bitwise, 550-552
- precedence, 82-84
- true/false values, 81-82

<< shift operator, 548-550

OR (|), 550-551

pointer arithmetic, 202

- precedence, 67-68
- subexpressions, 69
- summary, 86-87

relational, 70-71

- evaluating, 77-79
- precedence, 79-80

shift, 548-550

sizeof(), 521-522

stringizing (#), 569-571

ternary, 85

unary operator (-), 62-65

unary operator (++), 62, 65, 264

XOR (^), 550-551

OR operator (|), 550-551

OR operator (||), 81

output

- controlling, 348, 351-354
- defining, 326

device independent programming, 326

formatted output, 354

- fprintf() function, 350-352, 357-358

- printf() function, 350-357

printf() function, 146, 350-357

- conversion specifiers, 151-153
 - escape sequences, 147-151
 - format strings, 146
 - puts() function, 155
 - screen output
 - fputc() function, 349
 - fputs() function, 349
 - putc() function, 349
 - putchar() function, 347-349
 - puts() function, 349-350
 - standard input/output files. See predefined streams
 - streams
 - binary streams, 327
 - defining, 326
 - equivalence of streams, 329
 - files, 326
 - input/output functions, 328-329
 - predefined streams, 327-328
 - text streams, 327
 - strings
 - gets() function, 228-232
 - printf() function, 227-228
 - puts() function, 226-227
 - scanf() function, 232-235
- P**
- parameters
 - functions
 - scope of, 288
 - writing, 100-103
 - receiving arguments, 102
 - parentheses (), 94
 - functions, passing arguments, 110
 - operator precedence, 68
 - pointer declaration, 368
 - passing
 - arguments as variables, 488
 - arrays
 - passing by reference, 489
 - to functions, 204-209
 - listing, 205-207
 - data types to functions, 493-495
 - functions, passing by reference, 489
 - pointers to functions, 488-492
 - reference, passing by, 489-492
 - value, passing by, 488-491
 - paths
 - backslash (\), 419
 - filenames, 419
 - forward slash (/), 419
 - percent sign (%), conversion specifiers, 147
 - performance, macros versus functions, 571
 - perror() function, 519-520
 - planning structured programming, 97-99
 - pointer constants, 201
 - pointers, 190-193
 - arithmetic operations, 202
 - arrays
 - allocating memory, 372
 - arrays of structures and, 264-266
 - char type, 372-375
 - declaring, 373
 - element storage, 195-198
 - example of, 372, 375-381, 386
 - initializing, 374
 - names as pointers, 194-195
 - passing to functions, 204-209, 374-375
 - sorting, 376
 - sorting keyboard input, 377-379
 - strings, 372
 - compared to array subscript notation, 203
 - comparisons, 201
 - creating, 188-189
 - dangers of uninitialized pointers, 202-203
 - data types, 193-194
 - declaring, 189-190, 262, 371
 - char type, 372
 - pointers to pointers, 362-363
 - decrementing, 199-201
 - defining, 259-261
 - dereferencing void pointers, 493-495
 - differencing, 201
 - formatted output, 424
 - functions
 - passing pointers to, 488-492
 - returning pointers, 499-501
 - head pointers, 397-398
 - incrementing, 198-200, 264-266
 - initializing, 190, 261-262, 400
 - linked lists
 - adding elements to beginning of lists, 398-399
 - adding elements to empty links, 399
 - adding elements to end of lists, 400-401
 - adding elements to existing links, 400

- adding elements to middle of lists, 401-402
- adding links, 398
- deleting elements from lists, 402-403, 414
- deleting links, 398
- elements, 397
- example of, 403-406
- head pointers, 397
- implementing, 406-414
- links, 397
- lists of characters, 407-414
- loops, 406
- modifying links, 398
- nodes, 397
- structures, 396
- types of, 396
- listing, 191-192
- malloc() function, 221
- memory addresses, 188
- multidimensional arrays, 363-365, 368, 371
 - determining size of, 366
 - pointer arithmetic, 367
- naming, 188
- pointer arithmetic, 198-199
- pointers to functions
 - calling different functions, 389-390
 - controlling sort order, 393-395
 - declaring, 386, 396
 - initializing, 387-395
 - passing pointers as, 391-392
- strings, 219-220, 259
- structures, 261
 - creating pointers to, 262-263
 - members, 259-260
- to functions, initializing, 396
- typecasts and, 538
- uninitialized, 202-203, 400
- variables, declaring, 497
- void, 492-496, 521, 538
- portability, 6
- postfix mode, 63
- pow() function, 508
- precedence
 - brackets ([]), 368
 - indirection operator (*), 386
 - logical operators, 82-84
 - operators, 67-69, 86-87
 - relational operators, 79-80
 - subexpressions, 69
- precision modifiers, scanf() function, 340-341
- precision specifier, printf() function, 352
- precision variables, 42
- predefined functions, 10
- predefined macros, 576
- predefined streams, 327-328
- prefix mode, 63
- preprocessor
 - #define directive, 574
 - function macros, 568-572
 - macros versus functions, 571-572
 - substitution macros, 567-568, 572
 - #elif directive, 573
 - #else directive, 573
 - #endif directive, 573
 - #if directive, 573
 - #if...#endif directive, 573-575
 - #include directive, 572
 - #undef directive, 576
 - directives
 - checks, 575-576
 - defining, 567
 - header files, multiple inclusions of, 575
- print_report() function, 150
- printf() function, 10, 27, 105, 201, 227-228, 274, 350, 355-357, 571
 - arguments, 154
 - calling, 154
 - conversion characters, 351-352
 - conversion specifiers, 151-153
 - escape sequences, 147-151, 353-354
 - field-width specifier, 353
 - flags, 353
 - format strings, 146
 - functions with variable numbers of arguments, 496
 - l modifier, 352
 - ll specifier, 352
 - precision specifier, 352
- printf() statements, 27, 177
- printing
 - extended ASCII characters, listing, 216-217
 - multidimensional array elements, 368
- processor registers, 289
- program control statements, 71-77
- Program Development Cycle, 8-12
- program statements, 26
 - printf(), 27
 - return, 27
 - scanf(), 27
- programming
 - device independent programming, 326
 - modular programming, 562-563
 - advantages of, 560
 - external variables, 565-566
 - header files, 561, 565

- main modules, 560-561, 565
 - secondary modules, 560-561, 564
 - steps of, 8
 - structured programming
 - advantages, 97
 - bugs, 97
 - functions, entries/exits, 108
 - hierarchical structure, 98
 - menus, 99
 - planning, 97-99
 - tasks and subtasks, 98
 - top-down approach, 99
 - programs
 - arrays, for statements, 120
 - C, creating programs via, 8
 - executing
 - controlling, 121
 - do...while loops, 136-142
 - do...while statements, 140
 - for statements, 121-130
 - loops, nesting, 141-142
 - while statements, 130-136
 - exiting, 320
 - for statements, 120
 - Hello World, 13
 - compilation errors, 15-17
 - compiling hello.c, 14-15
 - linker errors, 17
 - terminating, exit() function, 320-321
 - promoting types in expressions, 534-535
 - prototype functions, 94-96, 109
 - ptr arguments, 541
 - putc() function, 349, 430
 - putchar() function, 333, 347-349, 430
 - puts() function, 108, 226-227, 349-350
 - calling, 155
 - escape sequences, 155
 - listing, 227
- ## Q
- qsort() function, 523-529
 - quotation marks (""), format strings, 146
- ## R
- radians, trigonometric functions, 506
 - RAM (Random Access Memory)
 - addresses, 38, 188
 - memory storage space, allocating, 539
 - rand() function, 182
 - random.c, 180-182
 - random arrays, 182
 - random file access, 436-442
 - reading
 - data in disk files, 423
 - strings, 228
 - gets() function, 228-232
 - scanf() function, 232-235
 - realloc() function, allocating memory, 541-543
 - recursion, calling functions, 112-114
 - reference, passing by, 489-492
 - register keyword, 289-290, 590
 - register variables, 289-290
 - relational operators, 70-71
 - evaluating, 77-79
 - precedence, 79-80
 - remove() function, 445-446
 - rename() function, 446-447
 - renaming files, 446-447
 - reserved keywords list, 589-591
 - reserved words, 7
 - restrict, 590
 - return keyword, 106, 590
 - return statements, 27, 94-96, 106
 - return type functions, writing, 100
 - returning values, writing functions, 106-108
 - rewind() function, 437-439, 445
 - Ritchie, Dennis, 6
 - running programs, 29
- ## S
- scanf() function, 27, 156-159, 232-235, 345-347
 - arguments, 160-161, 338-339
 - conversion specifiers, 339-340
 - handling extra characters, 341-343
 - listing, 234
 - precision modifiers, 340-341
 - scanf() statements, 27, 177
 - scope
 - local scope, function parameters, 288
 - variables, 280-282
 - errors, 281
 - external variables, 283
 - screen output
 - character output functions
 - fputc() function, 349
 - putc() function, 349
 - putchar(), 347-349
 - formatted output functions
 - fprintf(), 350-352

- fprintf() function, 357-358
 - printf() function, 350-357
- string output functions
 - fputs(), 349
 - puts(), 349-350
- searching
 - functions, bsearch(), 521-529
- strings
 - first occurrence of characters, 468-469
 - first occurrence of characters in second strings, 470-471
 - first occurrence of strings within strings, 473-474
 - last occurrence of characters, 470
 - nonmatching characters, 471-472
 - strchr() function, 468-469
 - strcspn() function, 470-471
 - strpbrk() function, 473
 - strrchr() function, 470
 - strspn() function, 471-472
 - strstr() function, 473-474
- secondary modules, 560-561, 564
- seconds.c, 66
- semicolon (;), 130
 - function headers, 101
 - function prototypes, writing, 109
- sequences, trigraph, 161
- sequential file access, 436-442
- shift operators, 548-550
- short, 590
- short type pointers, 193
- signed, 590
- signed decimal integer, displaying, 151
- simple expressions, 60
- simplestruct.c, 244-245
- sin() function, 506
- single-dimensional arrays, 169-173
- sinh() function, 507
- size arguments, 431-432
- size_t data type, 456
- sizeof() operators, 521-522, 590
- sleep() function, 311
- sorting
 - arrays of pointers, 376
 - functions, qsort(), 523-529
 - pointers to functions, controlling sort order, 393-395
- source code
 - accuracy, 30
 - compiling, 9-11
 - creating, 9
 - editors, 9
 - external variables, scope, 283
 - hello.c, 14
 - linking, 10
 - modular programming, 562-563
 - advantages of, 560
 - external variables, 565-566
 - header files, 561, 565
 - main modules, 560-561, 565
 - secondary modules, 560-561, 564
 - white space, 58
- source file, naming, 9
- spaghetti code, 306
- sqr() function, 561
- sqrt() function, 507
- square() function, 110
- src arguments
 - memcpy() function, 546
 - memmove() function, 546
- stacks, 488
- standard input/output files. See predefined streams
- statements, 58
 - #define, 178
 - assignment, 231
 - nesting, 61
 - operator precedence, 67
 - blocks, 59-60, 72, 573
 - branching, goto statements, 304-306
 - break, 300-302, 313-314
 - case, switch statements, 319-320
 - compound, 59-60
 - continue, 302-304
 - do...while loops, 140
 - for statements
 - arrays, 120
 - executing programs, 121-126
 - nesting, 127-130
 - structure, 121
 - functions, 95, 106
 - goto, 304-306
 - if statements, 71-77, 111
 - iteration, 114
 - label, 306
 - nested statements
 - assignment statements, 61
 - for statements, 127-130
 - if statements, 77
 - while, 134-136
 - null, 59
 - printf(), 177
 - program control, if statements, 71-72
 - return, 94-96, 106
 - scanf(), 177

- switch, 312
 - break statements and, 302, 313-314
 - case statements, 319-320
 - exit() function, 318
 - expressions, 311
 - menu systems, executing, 314-317
 - while statements, 131-133
 - nesting, 134-136
 - structure, 130
 - white space, 58-59
- static keyword, 286, 289, 590
- static memory allocation, 220-221, 538
- static variables
 - automatic versus, 285-288
 - external static, 289
- stdarg.h header file, functions with variable numbers of arguments, 496
- stdaux streams, 327
- stderr streams, 327, 357-358
- stdin streams, 327
- stdio.h header, 150, 227
- stdlib.h header
 - bsearch() function, 521
 - multidimensional arrays, 182
- stdout streams, 327
- stdprn streams, 327
- storage classes (variables), 291
- storage space (memory), 38
 - allocating
 - calloc() function, 540-541
 - dynamic memory allocation, 538
 - freeing via free() function, 543-545
 - malloc() function, 539
 - realloc() function, 541-543
 - static memory allocation, 538
 - bitwise operators
 - bit fields, 552-554
 - complement operator (~), 552
 - logical operators, 550-551
 - shift operators, 548-550
 - strcat() function, 461-462
 - strchr() function, 468-469
 - strcmp() function, 464-466
 - strcpy() function, 254, 380, 457-459
 - strcspn function, 470-471
 - streams
 - binary, 418
 - binary streams, 327
 - defining, 326
 - disk files, 418
 - character input, 428-429
 - character output, 423, 430-431
 - closing files, 435-436
 - detecting EOF, 443-444
 - direct I/O, 431-434
 - direct output, 423
 - file buffering, 435-436
 - formatted input, 427-428
 - formatted output, 423-426
 - opening, 419-423
 - random file access, 436-442
 - reading/writing data, 423
 - sequential file access, 436-442
 - equivalence of streams, 329
 - files, defining, 326
 - input/output functions, 328-329
 - predefined streams, 327-328
 - stdaux, 327
 - stderr, 327, 357-358
 - stdin, 327
 - stdout, 327
 - stdprn, 327
 - text streams, 327, 418
 - types of, 327
 - strftime() function, 511-515
 - stringizing operator (#), 569-571
 - strings
 - arguments, functions with variable numbers of arguments, 496
 - arrays of
 - characters, 218-219
 - pointers, 372
 - bsearch() function, 526-529
 - character test functions, 477-481
 - comparing
 - entire strings, 464-466
 - partial strings, 466-468
 - strcat() function, 464
 - strcmp() function, 464-466
 - strncmp() function, 466-468
 - concatenating
 - strcat() function, 461-462
 - strncat() function, 462-464
 - converting to numbers
 - double type numeric values, 476-477
 - floating point numeric values, 476
 - integers, 475
 - long longs, 476
 - longs, 475

- copying
 - strcpy() function, 457-459
 - strncpy() function, 459-460
- defining, 213, 259
- displaying
 - printf() function, 227-228
 - puts() function, 226-227
- filenames, 420
- functions with variable numbers of arguments, 496
- length, determining, 456-457
- literal, 219
- lowercase/uppercase characters, ANSI support for, 481-483
- memory allocation, 221
 - at compilation, 220
 - malloc() function, 222-226
- output functions
 - fputs(), 349
 - puts(), 349-350
- pointers to, 219-220
- qsort() function, 526-529
- reading, 232
 - gets() function, 228-231
 - scanf() function, 233-235
- searching
 - first occurrence of characters, 468-469
 - first occurrence of characters in second strings, 470-471
 - first occurrence of strings within strings, 473-474
 - last occurrence of characters, 470
- nonmatching characters, 471-472
- strchr() function, 468-469
- strcspn() function, 470-471
- strpbrk() function, 473
- strrchr() function, 470
- strspn() function, 471-472
- strstr() function, 473-474
- times, converting to strings, 511
- uppercase/lowercase characters, ANSI support for, 481-483
- white space, 58
- strlen() function, 379, 456-457
- strncat() function, 462-464
- strncmp() function, 466-468
- strncpy() function, 459-460
- strpbrk() function, 473
- strrchr() function, 470
- strspn() function, 471-472
- strstr() function, 473-474
- struct keyword, 242, 245-246, 590
- structure member operator (.). See dot operator (.)
- structure point operator. See indirect membership operator (->)
- structured programming. See also modular programming
 - advantages, 97
 - arrays of pointers, 375
 - bugs, 97
 - functions, entries/exits, 108
 - hierarchical structure, 98
 - menus, 99
 - planning, 97-99
 - tasks/subtasks, 98
 - top-down approach, 99
- structures
 - arrays of structures, 252-256
 - initializing, 257-258
 - pointers and, 264-266
 - arrays within structures, 250-252
 - bit fields, 552-554
 - complex structures
 - arrays within structures, 250-252
 - structures within structures, 246-249
 - declaring, 242-245
 - defining, 241-243
 - FILE, 420
 - functions, passing structures as arguments to, 267-268
 - initializing, 256-258
 - instances, defining, 245
 - linked lists, 396
 - members, 242
 - accessing, 243-244
 - pointers as, 259-261
 - pointers
 - arrays of structures and, 264-266
 - as structure members, 259-261
 - to structures, 261-263
 - struct keyword, 245-246
 - structures within structures, 246-249
 - synonyms, creating for structures, 274-275
 - tags, 242
 - tm, 509-511
 - typedef keyword, 274-275
- subexpressions, precedence, 69
- subscripts, 120
 - arrays, 169, 203
 - multidimensional arrays, 173

substitution macros, 567-568, 572

subtasks/tasks, structured programming, 98

subtraction operator (-), 65

switch statements, 312, 590

- break statements and, 302, 313-314
- case statements, 319-320
- exit() function, 318
- expressions, 311
- menu systems, executing, 314-317

symbolic character constants, 215

symbolic constants, 48, 60

- conditional compilation, 573
- creating, 567-568, 572
- declaring arrays, 174
- #define directive, 49
- defining, 49
- defining with const keyword, 50-52
- errno.h header file, 518
- external variables, 283

synonyms, creating for structures, 274-275

syntax (functions), calling, 95-96

- functions
 - calling, 95-96

T

tags, 242

tan() function, 506

tanh() function, 507

tasks and subtasks, structured programming, 98

temporary files, 450-452

terminating null character, strncat() function, 462

terminating programs, 320-321

ternary operator, 85

test-mode streams, 327

text

- literal, 146
- text-mode files
 - character input, 428-429
 - character output, 423, 430-431
 - CR-LF, 418
 - detecting EOF, 443-444
 - formatted input, 427-428
 - formatted output, 423-426
 - streams, 327, 418

third() function, 110

Thompson, Ken, 6

time functions, 509

- asctime(), 511, 515
- clock(), 513-515
- ctime(), 511
- difftime(), 513-515
- localtime(), 510
- mktime(), 511
- strftime(), 511-515
- time(), 510, 515
- usage examples, 513-515

tm structures, 509-511

tmpnam() function, 450-452

tolower() macro, 481-483

top-down approach, structured programming, 99

toupper() macro, 481-483

trigonometric functions

- acos(), 506
- asin(), 506
- atan(), 506
- atan2(), 506
- cos(), 506
- sin(), 506
- tan(), 506

trigraph sequences, 161

- ANSI standards, 162
- codes, 162

true/false values (logical operators), 81-82

two-dimensional arrays, 173-176, 364

type conversions

- automatic
 - conversion by assignment, 535-536
 - type promotion in expressions, 534-535
- explicit conversions via typecasts, 536
 - arithmetic expressions, 536-537
 - pointers, 538

type names. See tags

typecasts

- arithmetic expressions, 536-537
- explicit type conversions
 - arithmetic expressions, 536-537
 - pointers, 538
 - void pointers, 493
- typedef keyword, 590
- structures, 274-275
- variables, 45

typographical errors, 30

U

unary.c, 63-64

unary increment operator (++), 62-65, 264

unary operator (-), 62-65

unbuffered character input functions, 330

unconditional jumps. See branching statements

ungetc() function, 335, 480

uninitialized pointers, 202-203, 400

union keyword, 271-274, 590

- unions
 - accessing members, 271
 - declaring, 269
 - defining, 268-269
 - initializing, 269
 - members, accessing, 269-271
 - union keyword, 271-274
 - UNIX
 - memory allocation, 539
 - source code editors, 9
 - strings, comparing, 466
 - unsigned decimal integer, 151, 590
 - uppercase/lowercase
 - characters, ANSI support for, 481-483
 - user-defined functions, 24, 27, 91-93
 - usleep() function, 311
- V**
- va arg() macro, 497
 - va end() macro, 497-499
 - va list macro, 497
 - va start() macro, 497-499
 - values
 - passing by, 488-491
 - returning, writing functions, 106-108
 - variables, 26, 39, 51-52
 - accessibility, 280-281
 - approximate range, 42
 - arguments, passing variables as, 488
 - arrays, 168
 - automatic variables versus static variables, 285-288
 - char data type, 214
 - character variables, 215-218
 - counter variables, 124
 - declaring, 44
 - displaying with printf() function, 151
 - external, 282
 - declaring, 283
 - external static, 289
 - extern keyword, 283-285
 - modular independence, 283
 - modular programming, 565-566
 - scope, 283
 - symbolic constants, 283
 - when to use, 283
 - external static, 289
 - floating, 537
 - floating-point, 41
 - global. See external variables
 - in functions, 94
 - initializing, 45-46
 - integer, 41
 - lifetime, 280
 - local, 94
 - blocks and, 291-293
 - creating, 285
 - main function, 290-291
 - static variables versus automatic variables, 285-288
 - writing functions, 103-105
 - memory addresses, 188
 - naming, 39-40
 - numeric, 40-44
 - floating-point, 41
 - integer, 41
 - register keyword, 290
 - void pointers, 492
 - pointers, 191-192
 - creating, 188-189
 - data types, 193-194
 - declaring, 189-190, 497
 - initializing, 190
 - precision, 42
 - register, 289-290
 - scope, 280-282
 - errors, 281
 - external variables, 283
 - static variables
 - automatic variables versus, 285-288
 - external static variables, 289
 - storage classes, choosing between, 291
 - structures
 - accessing members, 243-244
 - arrays of, 255
 - arrays of structures, 252-258, 264-266
 - arrays within structures, 250-252
 - complex structures, 246-252
 - declaring, 242-245
 - defining, 241-245
 - initializing, 256-258
 - members of, 242
 - passing as arguments to functions, 267-268
 - pointers and arrays of structures, 264-266
 - pointers as structure members, 259-261
 - pointers, creating to structures, 261-263
 - struct keyword, 245-246
 - structures within structures, 246-249
 - tags, 242
 - typedef keyword, 45, 274-275
 - unions. See also arguments
 - accessing members, 269-271
 - declaring, 269
 - defining, 268-269
 - initializing, 269

- union keyword, 271-274
- visibility, 280
- visibility, variables, 280
- void pointers, 492-496, 521, 538, 590
- void return type, 112
- volatile, 590

X - Y - Z

- XOR operator (^), 550-551

W

- while loops, 132, 150, 591
 - break statement, 300-302
 - continue statement, 302-304
 - copying files, 450
 - detecting EOF, 444
 - infinite loops, 307-310
 - linked lists, 412
 - partial string comparisons, 468
 - random file access, 442
 - strings, converting to double type numeric values, 477
- while statements, 131-133
 - nesting, 134-136
 - structure, 130
- white space, 58-59, 156
- Windows, source code editors, 9
- writing
 - data to disk files, 423
 - functions
 - arguments, 100-103
 - body, 103-108
 - headers, 100-103
 - local variables, 103-105
 - names, 100
 - parameters, 100-103
 - prototypes, 109
 - returning values, 106-108
 - return types, 100
 - statements, 106