

# Applied C and C++ Programming

Alice E. Fischer

David W. Eggert

*University of New Haven*

Michael J. Fischer

*Yale University*

August 2018

**Copyright ©2018**

**by Alice E. Fischer, David W. Eggert, and Michael J. Fischer**

All rights reserved. This manuscript may be used freely by teachers and students in classes at the University of New Haven and at Yale University. Otherwise, no part of it may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the authors.

# Contents

<b>I Introduction</b>	<b>1</b>
<b>1 Computers and Systems</b>	<b>3</b>
1.1 The Physical Computer . . . . .	3
<i>Fig. 1.1:</i> Basic architecture of a modern computer. . . . .	3
1.1.1 The Processor . . . . .	3
1.1.2 The Memory . . . . .	4
<i>Fig. 1.2:</i> Main memory in a byte-addressable machine. . . . .	5
<i>Fig. 1.3:</i> The memory hierarchy. . . . .	5
1.1.3 Input and Output Devices . . . . .	8
1.1.4 The Bus . . . . .	9
1.1.5 Networks . . . . .	9
<i>Fig. 1.4:</i> A local network in a student lab. . . . .	10
1.2 The Operating System . . . . .	10
1.3 Languages . . . . .	12
1.3.1 Machine Language . . . . .	12
1.3.2 Assembly Languages . . . . .	13
1.3.3 High-Level Languages . . . . .	13
1.4 What You Should Remember . . . . .	15
1.4.1 Major Concepts . . . . .	15
1.4.2 Vocabulary . . . . .	15
1.5 Using Pencil and Paper . . . . .	15
1.5.1 Self-Test Exercises . . . . .	15
1.5.2 Using Pencil and Paper . . . . .	15
<b>2 Programs and Programming</b>	<b>17</b>
2.1 What Is a Program? . . . . .	17
<i>Fig. 2.1:</i> A simple algorithm: Find an average. . . . .	17
2.1.1 The Simplest Program . . . . .	18
<i>Fig. 2.2:</i> A simple program: Hail and farewell. . . . .	18
2.2 The stages of program development. . . . .	19
2.2.1 Define the Problem . . . . .	19
<i>Fig. 2.3:</i> Problem specification: Find the average exam score. . . . .	20
2.2.2 Design a Test Plan . . . . .	20
<i>Fig. 2.4:</i> Test plan for exam average program. . . . .	20
2.2.3 Design a Solution . . . . .	21
<i>Fig. 2.5:</i> Pseudocode program sketch: Find an average. . . . .	21
2.3 The Development Environment . . . . .	21
2.3.1 The Text Editor . . . . .	21
<i>Fig. 2.6:</i> The stages of program translation. . . . .	22
2.3.2 The Translator . . . . .	22

2.3.3	Linkers . . . . .	23
2.4	Source Code Construction . . . . .	23
2.4.1	Create a Source File . . . . .	23
2.4.2	Coding. . . . .	24
	<i>Fig. 2.7:</i> A C program. . . . .	24
2.4.3	Translate the Program and Correct Errors . . . . .	25
2.5	Program Execution and Testing . . . . .	26
2.5.1	Execute the Program . . . . .	26
	<i>Fig. 2.8:</i> Memory for constants. . . . .	26
	<i>Fig. 2.9:</i> Testing a program. . . . .	26
	<i>Fig. 2.10:</i> Memory during program execution. . . . .	27
2.5.2	Test and Verify . . . . .	28
2.6	What You Should Remember . . . . .	28
2.6.1	Major Concepts . . . . .	28
2.6.2	The moral of this story. . . . .	29
2.6.3	Vocabulary . . . . .	29
2.7	Exercises . . . . .	30
2.7.1	Self-Test Exercises . . . . .	30
2.7.2	Using Pencil and Paper . . . . .	30
2.7.3	Using the Computer . . . . .	31
<b>3</b>	<b>The Core of C</b> . . . . .	<b>33</b>
3.1	The Process of Compilation . . . . .	33
	<i>Fig. 3.1:</i> The stages of compilation. . . . .	33
3.2	The Parts of a Program . . . . .	34
3.2.1	Terminology. . . . .	34
	<i>Fig. 3.2:</i> How to lay out a simple program. . . . .	34
3.2.2	The main() program. . . . .	35
	<i>Fig. 3.3:</i> Words in C. . . . .	35
3.3	An Overview of Variables, Constants, and Expressions . . . . .	36
3.3.1	Values and Storage Objects. . . . .	36
3.3.2	Variables . . . . .	36
	<i>Fig. 3.4:</i> Syntax for declarations. . . . .	37
	<i>Fig. 3.5:</i> Simple declarations. . . . .	38
3.3.3	Constants and Literals . . . . .	39
	<i>Fig. 3.6:</i> Using constants. . . . .	40
3.3.4	Names and Identifiers . . . . .	40
3.3.5	Arithmetic and Formulas . . . . .	41
3.4	Simple Input and Output . . . . .	42
3.4.1	Formats. . . . .	42
3.5	A Program with Calculations . . . . .	44
	<i>Fig. 3.7:</i> Grapefruits and gravity. . . . .	44
3.6	The Flow of Control . . . . .	46
	<i>Fig. 3.8:</i> A complete flow diagram of a simple program. . . . .	47
	<i>Fig. 3.9:</i> Diagram of the grapefruits and gravity program. . . . .	47
3.7	Asking Questions: Conditional Statements . . . . .	48
3.7.1	The Simple if Statement . . . . .	48
	<i>Fig. 3.10:</i> Asking a question. . . . .	48
3.7.2	The if...else Statement . . . . .	50
	<i>Fig. 3.11:</i> Testing the limits. . . . .	50
	<i>Fig. 3.12:</i> Flow diagram for Testing the limits. . . . .	50
3.7.3	Which if Should Be Used? . . . . .	53

3.7.4	Options for Syntax and Layout (Optional Topic) . . . . .	53
	<i>Fig. 3.13:</i> The if statement with and without braces. . . . .	53
3.8	Loops and Repetition . . . . .	54
3.8.1	A Counting Loop . . . . .	54
	<i>Fig. 3.14:</i> Countdown. . . . .	54
3.8.2	An Input Data Validation Loop . . . . .	56
	<i>Fig. 3.15:</i> Input validation using while. . . . .	56
3.9	An Application . . . . .	58
	<i>Fig. 3.16:</i> Problem specification: Gas prices. . . . .	58
	<i>Fig. 3.17:</i> Test plan: Gas prices. . . . .	59
	<i>Fig. 3.18:</i> Problem solution: Gas prices. . . . .	60
3.10	What You Should Remember . . . . .	61
3.10.1	Major Concepts . . . . .	61
3.10.2	Programming Style . . . . .	63
3.10.3	Sticky Points and Common Errors . . . . .	63
3.10.4	New and Revisited Vocabulary . . . . .	64
3.10.5	Where to Find More Information . . . . .	64
3.11	Exercises . . . . .	65
3.11.1	Self-Test Exercises . . . . .	65
3.11.2	Using Pencil and Paper . . . . .	66
3.11.3	Using the Computer . . . . .	68
	<i>Fig. 3.19:</i> Sketch for the heat transfer program. . . . .	68
	<i>Fig. 3.20:</i> Sketch for the temperature conversion program. . . . .	68
	<i>Fig. 3.21:</i> Problem specification: Gasket area. . . . .	70
	<i>Fig. 3.22:</i> Flow diagram for the gasket area program. . . . .	70

## II Computation 73

4	Expressions <span style="float: right;">75</span>	
4.1	Expressions and Parse Trees . . . . .	75
4.1.1	Operators and Arity . . . . .	75
4.1.2	Precedence and Parentheses . . . . .	76
	<i>Fig. 4.1:</i> Arithmetic operators. . . . .	76
4.1.3	Parsing and Parse Trees . . . . .	77
	<i>Fig. 4.2:</i> Applying precedence. . . . .	77
	<i>Fig. 4.3:</i> Applying associativity. . . . .	77
	<i>Fig. 4.4:</i> Applying the parsing rules. . . . .	78
4.2	Arithmetic, Assignment, and Combination Operators . . . . .	79
	<i>Fig. 4.5:</i> Assignment and arithmetic combinations. . . . .	79
	<i>Fig. 4.6:</i> Parse tree for an assignment combination. . . . .	79
	<i>Fig. 4.7:</i> Arithmetic combinations. . . . .	80
4.3	Increment and Decrement Operators . . . . .	81
	<i>Fig. 4.8:</i> Increment and decrement operators. . . . .	81
4.3.1	Parsing Increment and Decrement Operators . . . . .	81
	<i>Fig. 4.9:</i> Increment and decrement trees. . . . .	81
4.3.2	Prefix versus Postfix Operators . . . . .	81
4.3.3	Mixing increment or decrement with other operators. . . . .	82
	<i>Fig. 4.10:</i> Evaluating an increment expression. . . . .	82
4.4	The sizeof operator. . . . .	83
	<i>Fig. 4.11:</i> The size of things. . . . .	83
4.5	Relational Operators . . . . .	84

<i>Fig. 4.12:</i> Relational operators.	84
4.5.1 True and False	84
4.5.2 The semantics of comparison.	85
4.6 Logical Operators	86
4.6.1 Truth-valued operators.	86
<i>Fig. 4.13:</i> Truth table for the logical operators.	86
4.6.2 Parse Trees for Logical Operators	86
4.6.3 Lazy Evaluation	87
<i>Fig. 4.14:</i> Precedence of the logical operators.	87
<i>Fig. 4.15:</i> Parsing logical operators.	87
<i>Fig. 4.16:</i> Lazy truth table for logical-AND.	87
<i>Fig. 4.17:</i> Lazy evaluation of logical-AND.	87
<i>Fig. 4.18:</i> Lazy truth table for logical-OR.	87
<i>Fig. 4.19:</i> Lazy evaluation of logical-OR.	88
4.7 Integer Operations	89
4.7.1 Integer Division and Modulus	89
<i>Fig. 4.20:</i> The modulus operation is cyclic.	90
4.7.2 Applying Integer Division and Modulus	90
<i>Fig. 4.21:</i> Visualizing the modulus operator.	91
<i>Fig. 4.22:</i> Positional notation and base conversion.	91
<i>Fig. 4.23:</i> Number conversion.	91
<i>Fig. 4.24:</i> A flow diagram for the base conversion.	91
4.8 Techniques for Debugging	94
4.8.1 Using Assignments and Printouts to Debug	94
<i>Fig. 4.25:</i> Problem specification: Ice cream for the picnic.	94
<i>Fig. 4.26:</i> How much ice cream?	94
4.8.2 Case Study: Using a Parse Tree to Debug	96
<i>Fig. 4.27:</i> Problem specification: Computing resistance.	96
<i>Fig. 4.28:</i> Computing resistance.	98
<i>Fig. 4.29:</i> Finding an expression error.	99
<i>Fig. 4.30:</i> Parsing the corrected expression.	99
4.9 What You Should Remember	100
4.9.1 Major Concepts	100
4.9.2 Programming Style	101
4.9.3 New and Revisited Vocabulary	101
<i>Fig. 4.31:</i> Difficult aspects of C operators.	101
4.9.4 Sticky Points and Common Errors	101
4.9.5 Where to Find More Information	101
4.10 Exercises	102
4.10.1 Self-Test Exercises	102
4.10.2 Using Pencil and Paper	104
4.10.3 Using the Computer	105
<b>5 Using Functions and Libraries</b>	<b>109</b>
5.1 Libraries	109
5.1.1 Standard Libraries	110
5.1.2 Other Libraries	110
5.1.3 Using Libraries	110
<i>Fig. 5.1:</i> A library is like an iceberg: only a small part is visible.	110
<i>Fig. 5.2:</i> Form of a simple function prototype.	111
5.2 Function Calls	112
<i>Fig. 5.3:</i> Form of a simple function call.	112

<i>Fig. 5.4:</i>	Calling library functions. . . . .	112
5.2.1	Call Graphs . . . . .	114
<i>Fig. 5.5:</i>	Call graph for Figure 5.4: Calling library functions. . . . .	115
5.2.2	Math Library Functions . . . . .	115
<i>Fig. 5.6:</i>	Functions in the math library. . . . .	115
<i>Fig. 5.7:</i>	Rounding down: <code>floor()</code> . . . . .	116
<i>Fig. 5.8:</i>	Rounding up: <code>ceil()</code> . . . . .	116
<i>Fig. 5.9:</i>	Rounding to the nearest integer: <code>rint()</code> . . . . .	117
<i>Fig. 5.10:</i>	Truncation via assignment. . . . .	117
5.2.3	Other Important Library Functions . . . . .	117
<i>Fig. 5.11:</i>	A few functions in other libraries. . . . .	118
5.3	Programmer-Defined Functions . . . . .	118
5.3.1	Function syntax. . . . .	119
5.3.2	Defining Void→Void Functions . . . . .	119
<i>Fig. 5.12:</i>	Using void→void functions. . . . .	119
<i>Fig. 5.13:</i>	Printing a banner on your output. . . . .	119
<i>Fig. 5.14:</i>	A call graph for the beep program. . . . .	119
5.3.3	Returning Results from a Function . . . . .	122
5.3.4	Arguments and Parameters . . . . .	122
5.3.5	Defining a Double→Double Function . . . . .	123
<i>Fig. 5.15:</i>	The grapefruit returns. . . . .	123
<i>Fig. 5.16:</i>	Call graph for the grapefruit returns. . . . .	125
<i>Fig. 5.17:</i>	Flow diagram for the grapefruit returns. . . . .	125
<i>Fig. 5.18:</i>	Definition of the <code>drop()</code> function. . . . .	125
5.4	Organization of a Module . . . . .	127
<i>Fig. 5.19:</i>	Functions, prototypes, and calls. . . . .	127
<i>Fig. 5.20:</i>	Function call graph with two levels of calls. . . . .	127
<i>Fig. 5.21:</i>	A function is a black box. . . . .	129
5.5	Application: Numerical Integration by Summing Rectangles . . . . .	130
<i>Fig. 5.22:</i>	The area under a curve. . . . .	130
<i>Fig. 5.23:</i>	Integration by summing rectangles. . . . .	130
5.6	Functions with More Than One Parameter . . . . .	132
5.6.1	Function Syntax: Two Parameters . . . . .	132
<i>Fig. 5.24:</i>	Using functions with two parameters and return values. . . . .	133
<i>Fig. 5.25:</i>	Functions with two parameters and return values. . . . .	133
5.7	Application: Generating “Random” Numbers . . . . .	135
5.7.1	Pseudo-Random Numbers . . . . .	135
5.7.2	How Good Is the Standard Random Number Generator? . . . . .	136
<i>Fig. 5.26:</i>	Generating random numbers. . . . .	136
5.8	Application: A Guessing Game . . . . .	138
5.8.1	Strategy . . . . .	138
5.8.2	Playing the Game . . . . .	139
<i>Fig. 5.27:</i>	Halving the range. . . . .	139
<i>Fig. 5.28:</i>	Can you guess my number? . . . . .	139
<i>Fig. 5.29:</i>	Guessing a number. . . . .	139
5.9	What You Should Remember . . . . .	142
5.9.1	Major Concepts . . . . .	142
5.9.2	Local Libraries and Header Files . . . . .	143
5.9.3	The Order of the Parts of a Program . . . . .	144
5.9.4	Programming Style . . . . .	145
5.9.5	Sticky Points and Common Errors . . . . .	145
5.9.6	Where to Find More Information . . . . .	146

5.9.7	New and Revisited Vocabulary . . . . .	146
5.10	Exercises . . . . .	147
5.10.1	Self-Test Exercises . . . . .	147
5.10.2	Using Pencil and Paper . . . . .	148
5.10.3	Using the Computer . . . . .	149
<b>6</b>	<b>More Repetition and Decisions</b>	<b>153</b>
6.1	New Loops . . . . .	153
6.1.1	The <b>for</b> Loop . . . . .	153
	<i>Fig. 6.1:</i> The <b>for</b> statement is a shorthand form of a <b>while</b> statement. . . . .	153
	<i>Fig. 6.2:</i> A simple program with a <b>for</b> statement. . . . .	153
	<i>Fig. 6.3:</i> A flow diagram for the <b>for</b> statement. . . . .	154
	<i>Fig. 6.4:</i> Diagrams of corresponding <b>while</b> and <b>for</b> loops. . . . .	154
6.1.2	The <b>do...while</b> Loop . . . . .	156
	<i>Fig. 6.5:</i> A flow diagram for the <b>do...while</b> statement. . . . .	156
6.1.3	Other Control Statements . . . . .	156
	<i>Fig. 6.6:</i> Using <b>break</b> with <b>while</b> and <b>do</b> . . . . .	157
	<i>Fig. 6.7:</i> Using <b>continue</b> with <b>while</b> and <b>do</b> . . . . .	157
	<i>Fig. 6.8:</i> Using <b>continue</b> with <b>for</b> . . . . .	158
6.1.4	Defective Loops . . . . .	158
	<i>Fig. 6.9:</i> No update and no exit: Two defective loops. . . . .	159
6.2	Applications of Loops . . . . .	159
6.2.1	Sentinel Loops . . . . .	160
	<i>Fig. 6.10:</i> A cash register program. . . . .	160
6.2.2	Query Loops . . . . .	161
	<i>Fig. 6.11:</i> Form of a query loop. . . . .	161
	<i>Fig. 6.12:</i> Repeating a calculation. . . . .	162
	<i>Fig. 6.13:</i> The <b>work()</b> and <b>drop()</b> functions. . . . .	162
	<i>Fig. 6.14:</i> Flow diagram for repeating a process. . . . .	162
6.2.3	Counted Loops . . . . .	164
	<i>Fig. 6.15:</i> Summing a series. . . . .	164
6.2.4	Input Validation Loops . . . . .	166
	<i>Fig. 6.16:</i> Input validation using a <b>while</b> statement. . . . .	166
6.2.5	Nested Loops . . . . .	167
	<i>Fig. 6.17:</i> Printing a table with nested <b>for</b> loops. . . . .	167
	<i>Fig. 6.18:</i> Flow diagram for the multiplication table program. . . . .	167
6.2.6	Delay Loops . . . . .	169
	<i>Fig. 6.19:</i> Using a delay loop. . . . .	169
	<i>Fig. 6.20:</i> Delaying progress, the <b>delay()</b> function. . . . .	169
6.2.7	Flexible-Exit Loops . . . . .	171
	<i>Fig. 6.21:</i> A structured loop with <b>break</b> . . . . .	171
	<i>Fig. 6.22:</i> Input validation loop using a <b>for</b> statement. . . . .	171
6.2.8	Counted Sentinel Loops . . . . .	172
	<i>Fig. 6.23:</i> Skeleton of a counted sentinel loop. . . . .	172
	<i>Fig. 6.24:</i> Breaking out of a loop. . . . .	172
	<i>Fig. 6.25:</i> Avoiding a break-out. . . . .	172
6.2.9	Search Loops . . . . .	174
	<i>Fig. 6.26:</i> Skeleton of a search loop. . . . .	174
6.3	The <b>switch</b> Statement . . . . .	174
6.3.1	Syntax and Semantics . . . . .	174
	<i>Fig. 6.27:</i> Problem specification: Wire gauge adequacy evaluation. . . . .	175
	<i>Fig. 6.28:</i> Diagrams for a nested conditional and a corresponding <b>switch</b> . . . . .	176

6.3.2 A switch Application . . . . .	177
<i>Fig. 6.29:</i> Using a <code>switch</code> . . . . .	177
6.4 Search Loop Application: Guess My Number . . . . .	179
<i>Fig. 6.30:</i> Problem specification: Guess my number . . . . .	179
<i>Fig. 6.31:</i> An input-driven search loop. . . . .	179
<i>Fig. 6.32:</i> A counted sentinel loop. . . . .	179
6.5 What You Should Remember . . . . .	181
6.5.1 Major Concepts . . . . .	181
6.5.2 Programming Style . . . . .	182
6.5.3 Sticky Points and Common Errors . . . . .	182
6.5.4 Where to Find More Information . . . . .	183
6.5.5 New and Revisited Vocabulary . . . . .	183
6.6 Exercises . . . . .	184
6.6.1 Self-Test Exercises . . . . .	184
6.6.2 Using Pencil and Paper . . . . .	185
6.6.3 Using the Computer . . . . .	187

### III Basic Data Types 191

<b>7 Using Numeric Types</b> <span style="float: right;">193</span>	
7.1 Integer Types . . . . .	193
<i>Fig. 7.1:</i> Names for integer types. . . . .	193
7.1.1 Signed and Unsigned Integers . . . . .	194
7.1.2 Short and long integers. . . . .	194
<i>Fig. 7.2:</i> ISO C integer representations. . . . .	194
<i>Fig. 7.3:</i> Integer literals in base 10. . . . .	195
7.2 Floating-Point Types in C . . . . .	195
<i>Fig. 7.4:</i> IEEE floating-point types. . . . .	195
<i>Fig. 7.5:</i> Floating-point literal examples . . . . .	196
7.3 Reading and Writing Numbers . . . . .	196
7.3.1 Integer Input . . . . .	197
<i>Fig. 7.6:</i> Integer conversion specifications. . . . .	197
7.3.2 Integer Output . . . . .	197
<i>Fig. 7.7:</i> Incorrect conversions produce garbage output. . . . .	197
7.3.3 Floating-Point Input . . . . .	198
7.3.4 Floating-Point Output . . . . .	198
<i>Fig. 7.8:</i> Basic floating-point conversion specifications. . . . .	199
<i>Fig. 7.9:</i> The %f output conversion. . . . .	199
<i>Fig. 7.10:</i> The %e output conversion. . . . .	199
<i>Fig. 7.11:</i> Sometimes %g output looks like an integer. . . . .	199
<i>Fig. 7.12:</i> Sometimes %g looks like %e. . . . .	199
<i>Fig. 7.13:</i> For tiny numbers, %g looks like %e. . . . .	199
<i>Fig. 7.14:</i> Sometimes %g looks like %f. . . . .	199
<i>Fig. 7.15:</i> Output conversion specifiers. . . . .	200
7.3.5 One Number may Appear in Many Ways . . . . .	200
7.4 Mixing Types in Computations . . . . .	202
7.4.1 Basic Type Conversions . . . . .	202
<i>Fig. 7.16:</i> Converting a <code>float</code> to a <code>double</code> . . . . .	202
7.4.2 Type Casts and Coercions . . . . .	203
<i>Fig. 7.17:</i> Kinds of casts. . . . .	203
<i>Fig. 7.18:</i> Rounding and truncation. . . . .	204

<i>Fig. 7.19:</i> Type coercion.	205
7.4.3 Diagramming Conversions	206
<i>Fig. 7.20:</i> Diagramming a cast and a coercion.	206
<i>Fig. 7.21:</i> Expressions with casts and coercions.	206
<i>Fig. 7.22:</i> Problem specification: computing resistance.	207
7.4.4 Using Type Casts to Avoid Integer Division Problems	207
<i>Fig. 7.23:</i> Computing resistance.	208
7.5 The Trouble with Numbers	209
7.5.1 Overflow	210
<i>Fig. 7.24:</i> Overflow and wrap with a four bit signed integer.	210
<i>Fig. 7.25:</i> Computing $N!$ .	211
7.5.2 Underflow	213
<i>Fig. 7.26:</i> Floating-point underflow.	213
7.5.3 Orders of Magnitude	214
7.5.4 Not a Number	214
<i>Fig. 7.27:</i> Calculation order matters.	215
7.5.5 Representational Error	215
7.5.6 Making Meaningful Comparisons	216
<i>Fig. 7.28:</i> An approximate comparison.	216
<i>Fig. 7.29:</i> Comparing floats for equality.	216
7.5.7 Application: Cruise Control	216
<i>Fig. 7.30:</i> Problem specification: Cruise control.	216
<i>Fig. 7.31:</i> Cruise control.	216
7.6 What You Should Remember	219
7.6.1 Major Concepts	219
7.6.2 Sticky Points and Common Errors	221
<i>Fig. 7.32:</i> Casts and conversions in C.	221
7.6.3 Programming Style	222
7.6.4 New and Revisited Vocabulary	223
7.6.5 Where to Find More Information	224
7.7 Exercises	224
7.7.1 Self-Test Exercises	224
7.7.2 Pencil and Paper	227
7.7.3 Using the Computer	229
<b>8 Character Data</b>	<b>233</b>
8.1 Representation of Characters	233
<i>Fig. 8.1:</i> The case bit in ASCII.	233
8.1.1 Character Types in C	234
8.1.2 The Different Interpretations	234
<i>Fig. 8.2:</i> Character types.	234
8.1.3 Character Literals	234
<i>Fig. 8.3:</i> Writing character constants.	234
<i>Fig. 8.4:</i> Useful predefined escape sequences.	234
8.2 Input and Output with Characters	235
8.2.1 Character Input	235
8.2.2 Character Output	236
<i>Fig. 8.5:</i> Printing the ASCII codes.	236
8.2.3 Using the I/O Functions	237
<i>Fig. 8.6:</i> Character input and output.	237
8.2.4 Other ways to skip whitespace.	239
<i>Fig. 8.7:</i> A function for skipping whitespace.	239

8.3	Operations on Characters . . . . .	240
8.3.1	Characters Are Very Short Integers . . . . .	240
<i>Fig. 8.8:</i>	Character operations. . . . .	240
8.3.2	Assignment . . . . .	240
8.3.3	Comparing Characters . . . . .	241
8.3.4	Character Arithmetic . . . . .	241
8.3.5	Other Character Functions . . . . .	241
8.4	Character Application: An Improved Processing Loop . . . . .	242
<i>Fig. 8.9:</i>	Improving the workmaster. . . . .	242
<i>Fig. 8.10:</i>	Using characters in a switch. . . . .	242
8.5	What You Should Remember . . . . .	245
8.5.1	Major Concepts . . . . .	245
8.5.2	Programming Style . . . . .	246
8.5.3	Sticky Points and Common Errors . . . . .	246
8.5.4	New and Revisited Vocabulary . . . . .	246
8.6	Exercises . . . . .	246
8.6.1	Self-Test Exercises . . . . .	246
8.6.2	Using Pencil and Paper . . . . .	247
8.6.3	Using the Computer . . . . .	249
<b>9</b>	<b>Program Design</b> . . . . .	<b>253</b>
9.1	Modular Programs . . . . .	253
9.2	Communication Between Functions . . . . .	253
9.2.1	The Function Interface . . . . .	254
<i>Fig. 9.1:</i>	Information flow in <code>pow2()</code> . . . . .	254
9.2.2	Parameter Passing . . . . .	255
<i>Fig. 9.2:</i>	The run-time stack. . . . .	255
9.3	Parameter Type Checking . . . . .	256
<i>Fig. 9.3:</i>	The declared prototype controls all. . . . .	256
<i>Fig. 9.4:</i>	The importance of parameter order. . . . .	257
9.4	Data Modularity . . . . .	258
<i>Fig. 9.5:</i>	Gas models before global elimination. . . . .	259
<i>Fig. 9.6:</i>	A call graph for the CO gas program. . . . .	259
<i>Fig. 9.7:</i>	Functions for gas models before global elimination. . . . .	259
9.4.1	Scope and Visibility . . . . .	259
9.4.2	Global and Local Names . . . . .	261
<i>Fig. 9.8:</i>	Name scoping in gas models program, before global elimination. . . . .	261
9.4.3	Eliminating Global Variables . . . . .	263
<i>Fig. 9.9:</i>	Eliminating the global variable. . . . .	263
<i>Fig. 9.10:</i>	Functions for gas models after global elimination. . . . .	263
9.5	Program Design and Construction . . . . .	265
9.5.1	The Process . . . . .	265
<i>Fig. 9.11:</i>	Problem specification and test plan: fin temperature. . . . .	265
9.5.2	Applying the Process: Stepwise Development of a Program . . . . .	266
<i>Fig. 9.12:</i>	First draft of <code>main()</code> for the fin program, with a function stub. . . . .	267
<i>Fig. 9.13:</i>	Fin program: First draft of <code>print_table()</code> function. . . . .	269
<i>Fig. 9.14:</i>	Temperature along a cooling fin. . . . .	272
9.6	What You Should Remember . . . . .	272
9.6.1	Major Concepts . . . . .	272
9.6.2	Programming Style . . . . .	274
9.6.3	Sticky Points and Common Errors . . . . .	274
9.6.4	New and Revisited Vocabulary . . . . .	275

9.7 Exercises . . . . .	275
9.7.1 Self-Test Exercises . . . . .	275
9.7.2 Using Pencil and Paper . . . . .	277
9.7.3 Using the Computer . . . . .	278
<b>10 An Introduction to Arrays</b>	<b>283</b>
10.1 Arrays . . . . .	283
10.1.1 Array Declarations and Initializers . . . . .	284
<i>Fig. 10.1:</i> Declaring an array of five floats. . . . .	284
<i>Fig. 10.2:</i> An initialized array of short ints. . . . .	284
<i>Fig. 10.3:</i> Length and initializer options. . . . .	284
<i>Fig. 10.4:</i> Initializing character arrays. . . . .	285
10.1.2 The Size of an Array . . . . .	285
<i>Fig. 10.5:</i> The size of an array. . . . .	286
10.1.3 Accessing Arrays . . . . .	286
<i>Fig. 10.6:</i> Simple subscripts. . . . .	287
<i>Fig. 10.7:</i> Computed subscripts. . . . .	287
<i>Fig. 10.8:</i> Subscript demo, the magnitude of a vector in 3-space. . . . .	288
10.1.4 Subscript Out-of-Range Errors . . . . .	289
10.2 Using Arrays . . . . .	289
10.2.1 Array Input . . . . .	289
<i>Fig. 10.9:</i> Filling an array with data. . . . .	289
10.2.2 Walking on Memory . . . . .	290
<i>Fig. 10.10:</i> Walking on memory. . . . .	290
<i>Fig. 10.11:</i> Before and after walking on memory. . . . .	290
10.3 Parallel Arrays . . . . .	292
<i>Fig. 10.12:</i> Problem specifications: Exam averages. . . . .	292
<i>Fig. 10.13:</i> Using parallel arrays. . . . .	292
<i>Fig. 10.14:</i> Parallel arrays can represent a table. . . . .	292
10.4 Array Arguments and Parameters . . . . .	295
10.5 An Array Application: Prime Numbers . . . . .	296
<i>Fig. 10.15:</i> Problem specifications: A table of prime numbers. . . . .	296
<i>Fig. 10.16:</i> Calculating prime numbers. . . . .	296
<i>Fig. 10.17:</i> Functions for the prime number program. . . . .	298
10.6 Searching an Array . . . . .	300
<i>Fig. 10.18:</i> Problem specifications: Recording bill payments. . . . .	300
<i>Fig. 10.19:</i> Main program for sequential search. . . . .	300
<i>Fig. 10.20:</i> Input and output functions for parallel arrays. . . . .	303
<i>Fig. 10.21:</i> Sequential search of a table. . . . .	304
<i>Fig. 10.22:</i> Searching without a second return statement. . . . .	304
10.7 The Maximum Value in an Array . . . . .	306
<i>Fig. 10.23:</i> Who owes the most? Main program for finding the maximum. . . . .	306
<i>Fig. 10.24:</i> Finding the maximum value. . . . .	306
<i>Fig. 10.25:</i> The maximum algorithm, step by step. . . . .	306
10.8 Sorting by Selection . . . . .	307
<i>Fig. 10.26:</i> The selection sort algorithm, step by step. . . . .	307
<i>Fig. 10.27:</i> Problem specifications: Sorting the Billing Records . . . . .	309
10.8.1 The Main Program . . . . .	309
<i>Fig. 10.28:</i> Call chart for selection sort. . . . .	310
10.8.2 Developing the <code>sort_data()</code> Function . . . . .	310
<i>Fig. 10.29:</i> Main program for selection sort. . . . .	310
<i>Fig. 10.30:</i> Sorting by selecting the maximum. . . . .	311

<i>Fig. 10.31:</i> Input and output for selection sort. . . . .	311
10.9 What You Should Remember . . . . .	312
10.9.1 Major Concepts . . . . .	312
10.9.2 Programming Style . . . . .	313
10.9.3 Sticky Points and Common Errors . . . . .	314
10.9.4 Where to Find More Information . . . . .	315
10.9.5 New and Revisited Vocabulary . . . . .	315
10.10 Exercises . . . . .	316
10.10.1 Self-Test Exercises . . . . .	316
10.10.2 Using Pencil and Paper . . . . .	318
10.10.3 Using the Computer . . . . .	319
<b>11 An Introduction to Pointers</b> . . . . .	<b>323</b>
11.1 A First Look at Pointers . . . . .	323
11.1.1 Pointer Values Are Addresses . . . . .	323
<i>Fig. 11.1:</i> A pointer and its referent. . . . .	323
<i>Fig. 11.2:</i> Pointer diagrams. . . . .	323
<i>Fig. 11.3:</i> Declaring a pointer variable. . . . .	323
<i>Fig. 11.4:</i> Initializing a pointer variable. . . . .	324
11.1.2 Pointer Operations . . . . .	325
<i>Fig. 11.5:</i> Pointers in memory. . . . .	325
<i>Fig. 11.6:</i> Pointer operations. . . . .	326
11.2 Call by Value/Address . . . . .	327
11.2.1 Address Arguments . . . . .	328
11.2.2 Pointer Parameters . . . . .	328
<i>Fig. 11.7:</i> Call by value/address. . . . .	328
<i>Fig. 11.8:</i> A swap function with an error. . . . .	331
<i>Fig. 11.9:</i> Seeing the difference. . . . .	331
<i>Fig. 11.10:</i> A swap function that works. . . . .	331
11.2.3 A More Complex Example . . . . .	331
11.3 Application: Statistical Measures . . . . .	332
<i>Fig. 11.11:</i> Problem specifications: Statistics. . . . .	332
<i>Fig. 11.12:</i> Mean and standard deviation. . . . .	332
<i>Fig. 11.13:</i> Call graph for the mean and standard deviation program. . . . .	332
<i>Fig. 11.14:</i> The <code>get_data()</code> and <code>stats()</code> functions. . . . .	332
11.3.1 Summary: Returning Results from a Function . . . . .	336
11.4 What You Should Remember . . . . .	336
11.4.1 Major Concepts . . . . .	336
11.4.2 Programming Style . . . . .	337
11.4.3 Sticky Points and Common Errors . . . . .	337
11.4.4 Where to Find More Information . . . . .	338
11.4.5 New and Revisited Vocabulary . . . . .	338
11.5 Exercises . . . . .	338
11.5.1 Self-Test Exercises . . . . .	338
11.5.2 Using Pencil and Paper . . . . .	340
11.5.3 Using the Computer . . . . .	342

<b>IV Representing Data</b>	<b>345</b>
<b>12 Strings</b>	<b>347</b>
12.1 String Representation . . . . .	347
12.1.1 String Literals . . . . .	348
<i>Fig. 12.1:</i> String literals. . . . .	348
<i>Fig. 12.2:</i> Quotes and comments. . . . .	348
12.1.2 A String Is a Pointer to an Array . . . . .	349
<i>Fig. 12.3:</i> Three kinds of nothing. . . . .	349
<i>Fig. 12.4:</i> Implementation of a cstring variable. . . . .	349
<i>Fig. 12.5:</i> Selecting the appropriate answer. . . . .	349
12.1.3 Declare an Array to Get a String . . . . .	350
<i>Fig. 12.6:</i> An array of characters. . . . .	350
12.1.4 Array vs. String . . . . .	351
<i>Fig. 12.7:</i> Array vs. string. . . . .	351
<i>Fig. 12.8:</i> A character array and a string. . . . .	351
<i>Fig. 12.9:</i> C++ strings. . . . .	351
12.1.5 C++ Strings . . . . .	352
12.2 C String I/O . . . . .	352
<i>Fig. 12.10:</i> String literals and string output in C. . . . .	352
<i>Fig. 12.11:</i> String literals and string output in C++. . . . .	352
12.2.1 String Output . . . . .	352
12.2.2 String Input in C . . . . .	356
<i>Fig. 12.12:</i> You cannot store an input string in a pointer. . . . .	356
<i>Fig. 12.13:</i> You need an array to store an input string. . . . .	356
12.2.3 String Input in C++ . . . . .	358
12.2.4 Guidance on Input . . . . .	359
12.3 String Functions . . . . .	359
12.3.1 Strings as Parameters . . . . .	359
<i>Fig. 12.14:</i> A string parameter in C. . . . .	359
<i>Fig. 12.15:</i> A string parameter in C++. . . . .	359
12.3.2 The String Library . . . . .	360
<i>Fig. 12.16:</i> The size of a C string. . . . .	362
<i>Fig. 12.17:</i> The size of a C++ string. . . . .	362
<i>Fig. 12.18:</i> Do not use == with C strings (but it works in C++). . . . .	363
<i>Fig. 12.19:</i> Computing the length of a string in C. . . . .	363
<i>Fig. 12.20:</i> Possible implementation of <code>strcmp()</code> . . . . .	363
<i>Fig. 12.21:</i> Searching for a character or a substring in C. . . . .	365
<i>Fig. 12.22:</i> Search, copy and concatenate in C. . . . .	366
<i>Fig. 12.23:</i> Search, copy and concatenate in C++. . . . .	366
12.4 Arrays of Strings . . . . .	367
<i>Fig. 12.24:</i> A ragged array. . . . .	367
12.4.1 The Menu Data Structure . . . . .	368
12.4.2 An Example: Selling Ice Cream . . . . .	368
<i>Fig. 12.25:</i> Selecting ice cream from a menu. . . . .	368
<i>Fig. 12.26:</i> A menu-handling function. . . . .	369
<i>Fig. 12.27:</i> Buying a Cone in C++. . . . .	369
<i>Fig. 12.28:</i> Using an invalid subscript. . . . .	370
12.5 String Processing Applications . . . . .	373
12.5.1 Password Validation . . . . .	373
<i>Fig. 12.29:</i> Password validation: Comparing strings. . . . .	373
<i>Fig. 12.30:</i> Password validation: Comparing strings. . . . .	373

12.5.2	The <code>menu_c()</code> Function . . . . .	375
	<i>Fig. 12.31:</i> The <code>menu_c()</code> function. . . . .	376
	<i>Fig. 12.32:</i> <code>menu_c</code> in C++ . . . . .	376
12.5.3	Menu Processing and String Parsing . . . . .	377
	<i>Fig. 12.33:</i> Flow diagram for Figure 12.34. . . . .	377
	<i>Fig. 12.34:</i> Magic memo maker in C. . . . .	377
	<i>Fig. 12.35:</i> Magic memo maker in C++ . . . . .	377
	<i>Fig. 12.36:</i> Using a string variable with a menu. . . . .	378
	<i>Fig. 12.37:</i> Composing and printing the memo in C. . . . .	382
	<i>Fig. 12.38:</i> Composing and printing the memo in C++ . . . . .	382
12.6	What You Should Remember . . . . .	384
12.6.1	Major Concepts . . . . .	384
12.6.2	Programming Style . . . . .	385
12.6.3	Sticky Points and Common Errors . . . . .	385
12.6.4	New and Revisited Vocabulary . . . . .	386
12.6.5	Where to Find More Information . . . . .	387
<b>13</b>	<b>Enumerated and Structured Types</b>	<b>389</b>
13.1	Enumerated Types . . . . .	389
13.1.1	Enumerations . . . . .	389
	<i>Fig. 13.1:</i> The form of an <code>enum</code> declaration in C and C++. . . . .	390
	<i>Fig. 13.2:</i> Four enumerations in C and again in C++. . . . .	390
	<i>Fig. 13.3:</i> Using an enumeration to name errors. . . . .	391
13.1.2	Printing Enumeration Codes . . . . .	391
13.1.3	Returning Multiple Function Results . . . . .	392
	<i>Fig. 13.4:</i> Memory for the quadratic root program. . . . .	392
	<i>Fig. 13.5:</i> Solving a quadratic equation in C. . . . .	392
	<i>Fig. 13.6:</i> Solving a quadratic equation in C++ . . . . .	394
	<i>Fig. 13.7:</i> The quadratic root function for both C and C++. . . . .	394
13.2	Structures in C . . . . .	397
	<i>Fig. 13.8:</i> Modern syntax for a C <code>struct</code> declaration. . . . .	397
	<i>Fig. 13.9:</i> A <code>struct</code> type declaration in C. . . . .	398
	<i>Fig. 13.10:</i> Declaring and initializing a structure. . . . .	398
13.3	Operations on Structures . . . . .	399
13.3.1	Structure Operations . . . . .	399
	<i>Fig. 13.11:</i> Set and use a pointer to a structure. . . . .	400
	<i>Fig. 13.12:</i> Access one member of a structure. . . . .	400
	<i>Fig. 13.13:</i> Structure assignment. . . . .	400
13.3.2	Using Structures with Functions . . . . .	400
	<i>Fig. 13.14:</i> Returning a structure from a function. . . . .	400
	<i>Fig. 13.15:</i> Call by value with a structure. . . . .	401
	<i>Fig. 13.16:</i> Call by address with a structure. . . . .	402
	<i>Fig. 13.17:</i> An array of structures. . . . .	403
13.3.3	Arrays of Structures . . . . .	403
	<i>Fig. 13.18:</i> Comparing two structures in C. . . . .	405
	<i>Fig. 13.19:</i> The <code>print_inventory</code> function. . . . .	405
	<i>Fig. 13.20:</i> Declarations for the lumber program. . . . .	405
	<i>Fig. 13.21:</i> Structure operations: the whole program in C. . . . .	405
13.3.4	Comparing Two Structures . . . . .	405
13.3.5	Putting the Pieces Together . . . . .	406
13.4	Structures and Classes in C++ . . . . .	408
13.4.1	Definitions and Conventions. . . . .	408

13.4.2 Function Members of a Class . . . . .	409
13.4.3 Encapsulation . . . . .	410
13.4.4 Instantiation and Memory Management . . . . .	411
13.5 The Lumber Program in C++ . . . . .	411
<i>Fig. 13.22:</i> The LumberT Class in C++ . . . . .	412
<i>Fig. 13.23:</i> The Lumber program in C++ . . . . .	412
<i>Fig. 13.24:</i> The Lumber functions in C++ . . . . .	414
13.6 Application: Points in a Rectangle . . . . .	416
<i>Fig. 13.25:</i> A rectangle on the $xy$ -plane. . . . .	416
<i>Fig. 13.26:</i> Problem specifications: Points in a rectangle. . . . .	416
<i>Fig. 13.27:</i> The test plan for points in a rectangle. . . . .	417
<i>Fig. 13.28:</i> Two structured types. . . . .	417
<i>Fig. 13.29:</i> Header file for points in a rectangle. . . . .	417
13.6.1 The program: Points in a Rectangle . . . . .	418
<i>Fig. 13.30:</i> Main program for points in a rectangle. . . . .	420
<i>Fig. 13.31:</i> Making a rectangle. . . . .	421
<i>Fig. 13.32:</i> Testing points. . . . .	422
<i>Fig. 13.33:</i> Comparing two points. . . . .	423
<i>Fig. 13.34:</i> Point location. . . . .	423
<i>Fig. 13.35:</i> Main function in C++ for points and rectangles. . . . .	425
13.7 Points in a Rectangle written in C++ . . . . .	425
<i>Fig. 13.36:</i> The C++ class declarations for points and rectangles. . . . .	427
<i>Fig. 13.37:</i> C++ functions for points. . . . .	428
<i>Fig. 13.38:</i> C++ functions for rectangles. . . . .	428
13.8 What You Should Remember . . . . .	430
13.8.1 Major Concepts . . . . .	430
13.8.2 Programming Style . . . . .	431
13.8.3 Sticky Points and Common Errors . . . . .	432
13.8.4 New and Revisited Vocabulary . . . . .	433
13.8.5 Where to Find More Information . . . . .	433
<b>14 Streams and Files</b>	<b>435</b>
14.1 Streams and Buffers . . . . .	435
14.1.1 Stream I/O . . . . .	435
<i>Fig. 14.1:</i> A stream carries data. . . . .	436
<i>Fig. 14.2:</i> Standard streams. . . . .	436
<i>Fig. 14.3:</i> The three default streams. . . . .	436
14.1.2 Buffering . . . . .	437
<i>Fig. 14.4:</i> An input buffer is a window on the data. . . . .	437
14.1.3 Formatted and Unformatted I/O. . . . .	438
14.2 Programmer-Defined Streams . . . . .	439
14.2.1 Defining and Using Streams . . . . .	439
<i>Fig. 14.5:</i> Opening streams in C++. . . . .	439
<i>Fig. 14.6:</i> Programmer-defined streams. . . . .	440
14.2.2 File Management Errors . . . . .	441
14.3 Stream Output . . . . .	441
<i>Fig. 14.7:</i> Specifications: Creating a data table. . . . .	441
<i>Fig. 14.8:</i> Writing a file of voltages. . . . .	441
14.4 Stream Input . . . . .	443
14.4.1 Input Functions . . . . .	443
14.4.2 Detecting End of File . . . . .	444
14.4.3 Reading Data from a File . . . . .	445

<i>Fig. 14.9:</i> Reading a data file. . . . .	445
14.4.4 Using Unformatted I/O . . . . .	447
<i>Fig. 14.10:</i> Copying a file. . . . .	447
14.4.5 Reading an Entire File at Once. . . . .	448
<i>Fig. 14.11:</i> Reading an entire file. . . . .	448
14.5 Stream Errors . . . . .	449
14.5.1 A Missing Newline . . . . .	450
14.5.2 An Illegal Input Character . . . . .	450
14.6 File Application: Random Selection Without Replacement . . . . .	451
<i>Fig. 14.12:</i> Problem specifications: A quiz. . . . .	452
<i>Fig. 14.13:</i> An elemental quiz. . . . .	452
<i>Fig. 14.14:</i> Call chart for the Element Quiz . . . . .	452
<i>Fig. 14.15:</i> The Quiz class declaration. . . . .	453
<i>Fig. 14.16:</i> The Element class declaration. . . . .	454
<i>Fig. 14.17:</i> The implementation of the Element class. . . . .	455
<i>Fig. 14.18:</i> The Quiz Constructor. . . . .	456
<i>Fig. 14.19:</i> The doQuiz function. . . . .	458
<i>Fig. 14.20:</i> The array after one question. . . . .	460
<i>Fig. 14.21:</i> The array after three questions. . . . .	460
<i>Fig. 14.22:</i> The array after five questions. . . . .	460
<i>Fig. 14.23:</i> The array after seven questions. . . . .	460
14.7 What You Should Remember . . . . .	461
14.7.1 Major Concepts . . . . .	461
14.7.2 Programming Style . . . . .	463
14.7.3 Sticky Points and Common Errors . . . . .	463
14.7.4 New and Revisited Vocabulary . . . . .	464
14.7.5 Where to Find More Information . . . . .	464
<b>15 Calculating with Bits</b>	<b>465</b>
15.1 Number Representation and Conversion . . . . .	465
15.1.1 Hexadecimal Notation . . . . .	465
<i>Fig. 15.1:</i> Hexadecimal numeric literals. . . . .	466
15.1.2 Number Systems and Number Representation . . . . .	466
<i>Fig. 15.2:</i> Place values. . . . .	466
15.1.3 Signed and Unsigned Integers . . . . .	467
<i>Fig. 15.3:</i> Two's complement representation of integers. . . . .	467
15.1.4 Representation of Real Numbers . . . . .	468
<i>Fig. 15.4:</i> Binary representation of reals. . . . .	468
15.1.5 Base Conversion . . . . .	469
<i>Fig. 15.5:</i> Converting binary numbers to decimal. . . . .	469
<i>Fig. 15.6:</i> Converting between hexadecimal and binary. . . . .	469
<i>Fig. 15.7:</i> Converting decimal numbers to binary. . . . .	469
<i>Fig. 15.8:</i> Converting hexadecimal numbers to decimal. . . . .	469
15.2 Hexadecimal Input and Output . . . . .	471
<i>Fig. 15.9:</i> Hexadecimal character literals. . . . .	471
<i>Fig. 15.10:</i> I/O for hex and decimal integers. . . . .	471
15.3 Bitwise Operators . . . . .	472
<i>Fig. 15.11:</i> Bitwise operators. . . . .	472
15.3.1 Masks and Masking . . . . .	473
<i>Fig. 15.12:</i> Bit masks for encrypting a number. . . . .	473
<i>Fig. 15.13:</i> Truth tables for bitwise operators. . . . .	473
<i>Fig. 15.14:</i> Bitwise AND (&). . . . .	474

<i>Fig. 15.15:</i> Bitwise OR ( ) . . . . .	474
<i>Fig. 15.16:</i> Bitwise XOR ( $\wedge$ ) . . . . .	474
<i>Fig. 15.17:</i> Just say no. . . . .	474
15.3.2 Shift Operators . . . . .	475
<i>Fig. 15.18:</i> Left and right shifts. . . . .	475
15.3.3 Example: Shifting and Masking an Internet Address . . . . .	476
<i>Fig. 15.19:</i> Problem specifications: Decoding an Internet address. . . . .	476
<i>Fig. 15.20:</i> Decoding an Internet address. . . . .	476
15.4 Application: Simple Encryption and Decryption . . . . .	477
<i>Fig. 15.21:</i> Problem specifications: Encrypting and decrypting a number. . . . .	478
<i>Fig. 15.22:</i> Encrypting a number. . . . .	478
<i>Fig. 15.23:</i> Decrypting a number. . . . .	478
15.5 Bitfield Types . . . . .	481
<i>Fig. 15.24:</i> Bitfield-structure demonstration. . . . .	482
15.5.1 Bitfield Application: A Device Controller (Advanced Topic) . . . . .	484
<i>Fig. 15.25:</i> Problem specifications: Skylight controller. . . . .	484
<i>Fig. 15.26:</i> Declarations for the skylight controller: <code>skylight.hpp</code> . . . . .	486
<i>Fig. 15.27:</i> A skylight controller. . . . .	486
<i>Fig. 15.28:</i> Operations for the skylight controller. . . . .	488
15.6 What You Should Remember . . . . .	491
15.6.1 Major Concepts . . . . .	491
<i>Fig. 15.29:</i> Summary of bitwise operators. . . . .	491
15.6.2 Programming Style . . . . .	492
15.6.3 Sticky Points and Common Errors . . . . .	492
15.6.4 New and Revisited Vocabulary . . . . .	493
15.7 Exercises . . . . .	493
15.7.1 Self-Test Exercises . . . . .	493
15.7.2 Using Pencil and Paper . . . . .	494
15.7.3 Using the Computer . . . . .	495
<i>Fig. 15.30:</i> Problem 10. . . . .	499
<b>V Pointers</b>	<b>501</b>
<b>16 Dynamic Arrays</b>	<b>503</b>
16.1 Operator Definitions . . . . .	503
16.2 Pointers—Old and New Ideas . . . . .	503
16.2.1 Pointer Declarations and Initialization . . . . .	504
<i>Fig. 16.1:</i> Pointing at an array. . . . .	504
16.2.2 Using Pointers . . . . .	504
<i>Fig. 16.2:</i> Indirect reference. . . . .	504
<i>Fig. 16.3:</i> Direct and indirect assignment. . . . .	505
<i>Fig. 16.4:</i> Input and output through pointers. . . . .	505
16.2.3 Pointer Arithmetic and Logic . . . . .	506
<i>Fig. 16.5:</i> Pointer addition and subtraction. . . . .	506
<i>Fig. 16.6:</i> Incrementing a pointer. . . . .	507
<i>Fig. 16.7:</i> Pointer comparisons. . . . .	507
16.2.4 Using Pointers with Arrays . . . . .	507
<i>Fig. 16.8:</i> An array with a cursor and a sentinel. . . . .	508
<i>Fig. 16.9:</i> An increment loop. . . . .	508
16.3 Dynamic Memory Allocation . . . . .	509
<i>Fig. 16.10:</i> Allocating new memory. . . . .	510

16.3.1 Simple Memory Allocation . . . . .	510
16.4 Arrays that Grow . . . . .	512
16.4.1 The Flex Class . . . . .	512
<i>Fig. 16.11:</i> Flex: a growing array . . . . .	512
<i>Fig. 16.12:</i> Flex functions. . . . .	512
16.5 Application: Insertion Sort . . . . .	515
<i>Fig. 16.13:</i> Inner loop of the Insertion sort . . . . .	515
<i>Fig. 16.14:</i> Insertion sort. . . . .	515
<i>Fig. 16.15:</i> Insertion sort using a dynamic array. . . . .	516
<i>Fig. 16.16:</i> Insertion sort. . . . .	516
<i>Fig. 16.17:</i> The Sorter class. . . . .	518
<i>Fig. 16.18:</i> Insertion sort using a Flex array. . . . .	519
16.6 Selection Sort . . . . .	521
<i>Fig. 16.19:</i> Selection sort using a dynamic array. . . . .	521
<i>Fig. 16.20:</i> The controller class: Charges. . . . .	521
<i>Fig. 16.21:</i> Functions for the Charges class. . . . .	523
<i>Fig. 16.22:</i> The data class: Transaction. . . . .	523
<i>Fig. 16.23:</i> Functions for the data class, Transaction. . . . .	523
16.7 What You Should Remember . . . . .	526
16.7.1 Major Concepts . . . . .	526
16.7.2 Programming Style . . . . .	527
16.7.3 Sticky Points and Common Errors . . . . .	527
16.7.4 Vocabulary . . . . .	528
16.7.5 Self-Test Exercises . . . . .	529
16.7.6 Using Pencil and Paper . . . . .	530
16.7.7 Using the Computer . . . . .	531
<i>Fig. 16.24:</i> Poker hands, high to low. . . . .	531
<b>17 Vectors and Iterators</b>	<b>533</b>
17.1 The Standard Template Library–STL . . . . .	533
17.1.1 Containers . . . . .	534
17.2 The vector Class . . . . .	534
<i>Fig. 17.1:</i> Vector operations. . . . .	534
<i>Fig. 17.2:</i> Vector iterators. . . . .	535
17.2.1 Using the STL vector Class . . . . .	535
<i>Fig. 17.3:</i> Vector demo program. . . . .	535
<i>Fig. 17.4:</i> Problem Specification. . . . .	537
17.3 A 3-D dynamic object. . . . .	538
<i>Fig. 17.5:</i> The Pleasant Lakes Club. . . . .	538
<i>Fig. 17.6:</i> The FamilyT class. . . . .	540
<i>Fig. 17.7:</i> The FamilyT functions. . . . .	540
17.4 Using Vectors: A Simulation . . . . .	543
17.4.1 Transient Heat Conduction in a Semi-Infinite Slab . . . . .	543
<i>Fig. 17.8:</i> Heat conduction in a semi-infinite slab. . . . .	543
17.4.2 Simulating the Cooling Process . . . . .	544
<i>Fig. 17.9:</i> A call chart for the heat flow simulation. . . . .	544
<i>Fig. 17.10:</i> A heat flow simulation. . . . .	544
<i>Fig. 17.11:</i> The Slab class. . . . .	544
<i>Fig. 17.12:</i> The Slab constructor and destructor. . . . .	545
<i>Fig. 17.13:</i> <code>simCool()</code> : doing the simulation. . . . .	547
<i>Fig. 17.14:</i> <code>Print()</code> , <code>results()</code> , and <code>debug()</code> . . . . .	548
17.5 What You Should Remember . . . . .	550

17.5.1 Major Concepts . . . . .	550
17.5.2 Programming Style . . . . .	550
17.5.3 Sticky Points and Common Errors . . . . .	551
17.5.4 New and Revisited Vocabulary . . . . .	551
17.6 Exercises . . . . .	551
17.6.1 Self-Test Exercises . . . . .	551
17.6.2 Using Pencil and Paper . . . . .	552
17.6.3 Using the Computer . . . . .	552
<b>18 Array Data Structures</b>	<b>555</b>
18.1 Concepts . . . . .	555
18.1.1 Declarations and Memory Layout . . . . .	555
<i>Fig. 18.1:</i> A two-dimensional array and initializer. . . . .	555
<i>Fig. 18.2:</i> Layout of an array in memory. . . . .	555
18.1.2 Using <code>typedef</code> for Two-Dimensional Arrays . . . . .	556
<i>Fig. 18.3:</i> Using <code>typedef</code> for 2D arrays. . . . .	556
18.1.3 Ragged Arrays . . . . .	556
<i>Fig. 18.4:</i> A menu function. . . . .	556
18.1.4 A Matrix . . . . .	558
<i>Fig. 18.5:</i> Travel time for a two-day trip. . . . .	558
18.1.5 An Array of Arrays . . . . .	560
<i>Fig. 18.6:</i> Declaring an array of arrays. . . . .	561
<i>Fig. 18.7:</i> Calculating wind speed. . . . .	561
<i>Fig. 18.8:</i> Printing the wind speed table. . . . .	561
18.1.6 Dynamic Matrix: An Array of Pointers . . . . .	563
18.1.7 Multidimensional Arrays . . . . .	564
<i>Fig. 18.9:</i> An array of dynamic arrays. . . . .	564
<i>Fig. 18.10:</i> A three-dimensional array. . . . .	564
<i>Fig. 18.11:</i> Problem specifications: 2D or 3D point transformation. . . . .	564
18.2 Application: Transformation of 2D Point Coordinates . . . . .	564
<i>Fig. 18.12:</i> 2D point transformation—main program. . . . .	566
<i>Fig. 18.13:</i> Three Class Declarations . . . . .	566
<i>Fig. 18.14:</i> Functions for the Transform class. . . . .	568
<i>Fig. 18.15:</i> Functions for the Drawing class. . . . .	570
18.3 Application: Image Processing . . . . .	570
18.3.1 Digital images. . . . .	570
<i>Fig. 18.16:</i> Problem specifications: Image smoothing. . . . .	572
18.3.2 Smoothing an image. . . . .	572
<i>Fig. 18.17:</i> Image smoothing—main program. . . . .	572
<i>Fig. 18.18:</i> Getting parameters for the smoothing. . . . .	573
<i>Fig. 18.19:</i> Prevent accidental overwriting of existing file. . . . .	573
<i>Fig. 18.20:</i> The Pgm class. . . . .	576
<i>Fig. 18.21:</i> The Pgm constructors. . . . .	578
<i>Fig. 18.22:</i> Parsing and printing the header. . . . .	580
<i>Fig. 18.23:</i> Writing the Pgm file. . . . .	581
<i>Fig. 18.24:</i> Smoothing. . . . .	581
<i>Fig. 18.25:</i> Results of image smoothing program. . . . .	583
18.4 Application: Gaussian Elimination . . . . .	583
18.4.1 An Implementation of Gauss's Algorithm . . . . .	584
<i>Fig. 18.26:</i> Solving a system of linear equations. . . . .	585
<i>Fig. 18.27:</i> A call chart for Gaussian elimination. . . . .	585
<i>Fig. 18.28:</i> The Equation class declarations. . . . .	586

<i>Fig. 18.29:</i>	The Gauss class declaration.	586
<i>Fig. 18.30:</i>	Constructing the model.	588
<i>Fig. 18.31:</i>	Output functions for the Gauss class.	589
<i>Fig. 18.32:</i>	Output for the Equation class.	590
<i>Fig. 18.33:</i>	The solve() function.	591
<i>Fig. 18.34:</i>	Finding the next pivot row.	591
<i>Fig. 18.35:</i>	<code>Equation::scale()</code> .	593
<i>Fig. 18.36:</i>	<code>Equation::wipe()</code> .	593
18.5	What You Should Remember	593
18.5.1	Major Concepts	593
18.5.2	Programming Style	594
18.5.3	Sticky Points and Common Errors	595
18.5.4	New and Revisited Vocabulary	595
18.6	Exercises	596
18.6.1	Self-Test Exercises	596
18.6.2	Using Pencil and Paper	598
18.6.3	Using the Computer	599
<b>VI</b>	<b>Developing Sophistication</b>	<b>605</b>
<b>19</b>	<b>Recursion</b>	<b>607</b>
19.1	Storage Classes	607
19.1.1	Automatic Storage	607
19.1.2	Static Storage	608
19.1.3	External Storage (Advanced Topic)	608
19.2	The Run-Time Stack (Advanced Topic)	609
19.2.1	Stack Frames	609
19.2.2	Stack Diagrams and Program Traces	610
	<i>Fig. 19.1:</i> The run-time stack.	610
	<i>Fig. 19.2:</i> The run-time stack for the gas pressure program.	611
19.3	Iteration and Recursion	611
19.3.1	The Nature of Iteration	611
19.3.2	The Nature of Recursion	612
	<i>Fig. 19.3:</i> A recursive program to find a minimum.	612
19.3.3	Tail Recursion	612
19.4	A Simple Example of Recursion	613
19.5	A More Complex Example: Binary Search	615
	<i>Fig. 19.4:</i> Call chart for binary search.	615
	<i>Fig. 19.5:</i> The binary search main program.	616
	<i>Fig. 19.6:</i> The Table class declaration.	616
	<i>Fig. 19.7:</i> The Table constructor.	617
	<i>Fig. 19.8:</i> Ensuring that the data is sorted.	618
	<i>Fig. 19.9:</i> Searching for numbers.	618
	<i>Fig. 19.10:</i> The binary search function.	620
	<i>Fig. 19.11:</i> Diagrams of a Binary Search: Ready to begin.	622
	<i>Fig. 19.12:</i> Second active call.	622
	<i>Fig. 19.13:</i> Third active call.	622
	<i>Fig. 19.14:</i> Fourth active call.	623
	<i>Fig. 19.15:</i> Call graph for the quicksort program.	623
19.6	Quicksort	623
	<i>Fig. 19.16:</i> The main program for <code>quicksort()</code> .	624

<i>Fig. 19.17:</i> Tester generates data for <code>quicksort()</code> . . . . .	624
<i>Fig. 19.18:</i> Implementation for the Tester class. . . . .	625
<i>Fig. 19.19:</i> The Quick class. . . . .	625
<i>Fig. 19.20:</i> The <code>quicksort()</code> function. . . . .	627
<i>Fig. 19.21:</i> <code>sortToCutoff ()</code> . . . . .	627
<i>Fig. 19.22:</i> Setting the pivot. . . . .	628
<i>Fig. 19.23:</i> A diagram of <code>setPivot()</code> . . . . .	628
<i>Fig. 19.24:</i> The <code>partition()</code> function, the heart of <code>quicksort()</code> . . . . .	630
<i>Fig. 19.25:</i> The first pass through <code>partition</code> . . . . .	631
<i>Fig. 19.26:</i> The first recursive call. . . . .	631
<i>Fig. 19.27:</i> Finishing the recursions. . . . .	631
<i>Fig. 19.28:</i> Finish sorting using insertion sort. . . . .	631
<i>Fig. 19.29:</i> Insertion sort, first part. . . . .	633
<i>Fig. 19.30:</i> Insertion sort, second part. . . . .	633
19.6.1 Possible Improvements . . . . .	635
19.7 What You Should Remember . . . . .	635
19.7.1 Major Concepts . . . . .	635
19.7.2 Programming Style . . . . .	638
19.7.3 Sticky Points and Common Errors . . . . .	638
19.7.4 New and Revisited Vocabulary . . . . .	639
19.8 Exercises . . . . .	639
19.8.1 Self-Test Exercises . . . . .	639
19.8.2 Using Pencil and Paper . . . . .	640
19.8.3 Using the Computer . . . . .	641
<b>20 Command Line Arguments</b>	<b>643</b>
20.1 Command-Line Arguments . . . . .	643
20.1.1 The Argument Vector . . . . .	643
<i>Fig. 20.1:</i> The argument vector. . . . .	643
20.1.2 Decoding Arguments . . . . .	643
<i>Fig. 20.2:</i> Using command-line arguments. . . . .	644
<i>Fig. 20.3:</i> The Sorter class. . . . .	645
<i>Fig. 20.4:</i> The Sorter Constructor. . . . .	645
<i>Fig. 20.5:</i> Reading the data. . . . .	648
<i>Fig. 20.6:</i> Printing the sorted data. . . . .	648
<i>Fig. 20.7:</i> Sort in ascending or descending order. . . . .	649
20.2 Functions with Variable Numbers of Arguments . . . . .	651
<i>Fig. 20.8:</i> The <code>fatal()</code> function. . . . .	651
20.3 Modular Organization . . . . .	652
20.3.1 File Management Issues with Modular Construction . . . . .	652
20.3.2 Building a Multimodule Program . . . . .	654
20.3.3 Using a makefile. . . . .	655
<i>Fig. 20.9:</i> Steps in building a program. . . . .	656
20.3.4 A Unix makefile. . . . .	657
<i>Fig. 20.10:</i> A Unix makefile for <code>game</code> . . . . .	658
<b>A The ASCII Code</b>	<b>661</b>
<b>B The Precedence of Operators in C</b>	<b>663</b>
<i>Fig. B.1:</i> The precedence of operators in C. . . . .	663

<b>C Keywords</b>	<b>665</b>
C.1 Preprocessor Commands . . . . .	665
C.2 Control Words . . . . .	665
C.3 Types and Declarations . . . . .	665
C.4 Additional C++ Reserved Words . . . . .	665
C.5 An Alphabetical List of C and C++ Reserved Words . . . . .	666
<b>D Advanced Aspects C Operators</b>	<b>667</b>
D.1 Assignment Combination Operators . . . . .	667
<i>Fig. D.1:</i> Assignment combinations. . . . .	667
<i>Fig. D.2:</i> A parse tree for assignment combinations. . . . .	667
D.2 More on Lazy Evaluation and Skipping . . . . .	667
<i>Fig. D.3:</i> Lazy evaluation. . . . .	667
<i>Fig. D.4:</i> Skip the whole operand. . . . .	668
<i>Fig. D.5:</i> And nothing but the operand. . . . .	670
D.2.1 Evaluation Order and Side-Effect Operators . . . . .	670
D.3 The Conditional Operator . . . . .	670
<i>Fig. D.6:</i> A flowchart for the conditional operator. . . . .	671
<i>Fig. D.7:</i> A tree for the conditional operator. . . . .	671
D.4 The Comma Operator . . . . .	672
D.5 Summary . . . . .	672
<i>Fig. D.8:</i> Complications in use of side-effect operators. . . . .	673
<b>E Dynamic Allocation in C</b>	<b>675</b>
<i>Fig. E.1:</i> Dynamic memory allocation functions. . . . .	675
E.0.1 Mass Memory Allocation . . . . .	675
E.0.2 Cleared Memory Allocation . . . . .	677
E.0.3 Freeing Dynamic Memory . . . . .	677
E.0.4 Resizing an Array . . . . .	678
<b>F The Standard C Environment</b>	<b>681</b>
F.1 Built-in Facilities . . . . .	681
F.2 Standard Files of Constants . . . . .	681
<i>Fig. F.1:</i> Minimum values. . . . .	682
F.3 The Standard Libraries and <code>main()</code> . . . . .	682
F.3.1 The Function <code>main()</code> . . . . .	682
F.3.2 Characters and Conversions . . . . .	683
F.3.3 Mathematics . . . . .	683
F.3.4 Input and Output . . . . .	685
F.3.5 Standard Library . . . . .	687
F.3.6 Strings . . . . .	688
F.3.7 Time and Date . . . . .	690
F.3.8 Variable-Length Argument Lists . . . . .	691
F.4 Libraries Not Explored . . . . .	691

# **Part I**

## **Introduction**



# Chapter 1

# Computers and Systems

## 1.1 The Physical Computer

It is not necessary to understand how a computer works in order to use one—but it helps. An elementary understanding of computer architecture helps demystify the nature and rules of programming languages and enables one to use these languages more wisely. It is essential for anyone who needs to attach devices to a computer or buy one wisely. The architecture of modern computers varies greatly from type to type, and new developments happen every year. Therefore, it is impossible to describe how all computers work. The following discussion is intended to give a general idea of the elements common or universal to personal computers and workstations today.

The main logical parts of a computer, diagrammed in Figure 1.1, can be roughly compared to parts of a human body:

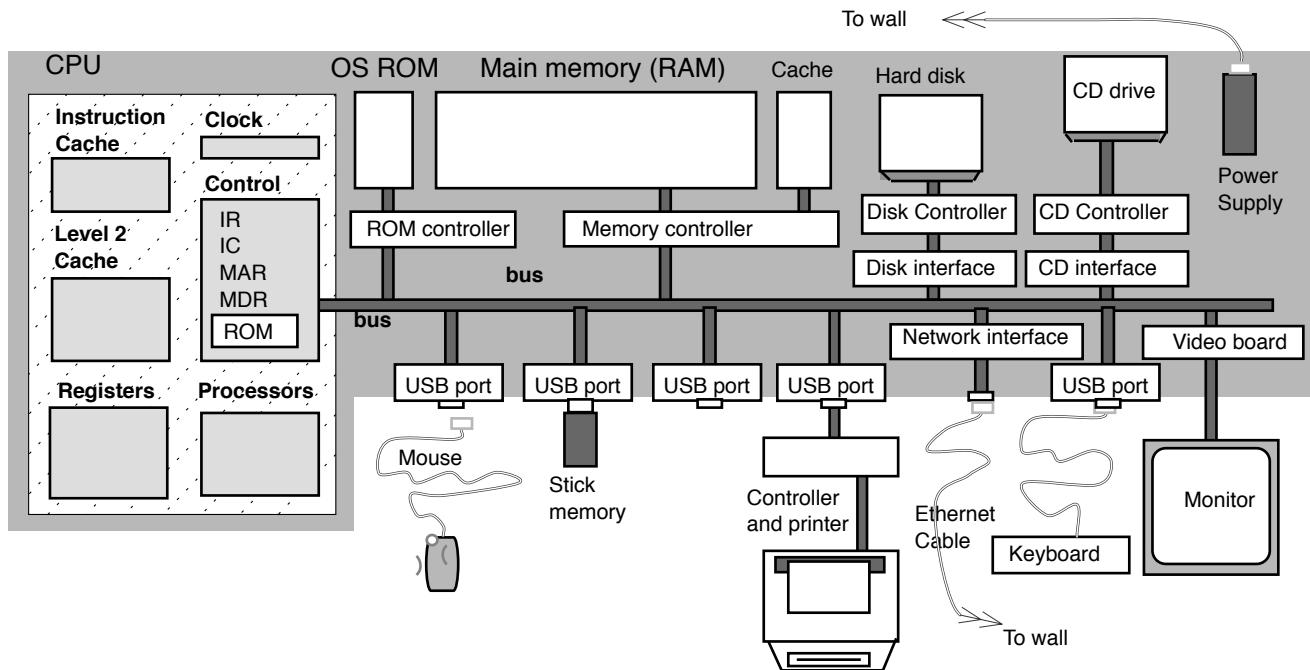
- The **CPU** (central processing unit) is the brain of the computer.
- The computer’s **RAM** (random access memory) chips are its memory.
- The **bus** is the nervous system; it carries information between the CPU and everything else in the computer.
- The **input** devices (e.g., keyboard) are the computer’s senses.
- The **output** devices (e.g., monitor) are the computer’s effectors (hands, voice).

### 1.1.1 The Processor

In a typical modern computer, the central processing unit (CPU) is the main element on the processor chip. The CPU controls and coordinates the whole machine. It contains a set of **registers**:

- The instruction register (IR) holds the current machine instruction.
- The instruction counter (IC) holds the address of the next machine instruction.
- The memory data register (MDR) holds the data currently in use.
- The memory address register (MAR) holds the address from which the data came.

One of its components is the **clock**, which ticks at a fixed rate and controls the fundamental speed at which all of the computer’s operations work. The clock rate on a microprocessor chip is set as fast as is (conservatively) possible without causing processing errors. This setting is the megahertz (MHz) rating published by the manufacturer.



**Figure 1.1. Basic architecture of a modern computer.**

**The ALU.** The *arithmetic and logic unit* (ALU) is the part of the processor containing the many circuits that actually perform computations. Typically, an ALU includes instructions for addition, negation, and multiplication of integers; comparison; logical operations; and other actions. Many computers also have a **floating-point coprocessor**, for handling arithmetic operations on real (floating-point) numbers. Floating-point instructions are important for many scientific applications to achieve adequate accuracy at an acceptable speed. Taken together, this set of instructions forms the **machine language** for that particular processor.

**Control ROM and the instruction cycle.** A small read-only memory inside the control unit contains instructions (called *microcode*) that control all parts of the CPU and define the actions of the instruction cycle, the ALU, and the instruction cache (discussed next).

To use a computer, we write a **program**, which is a series of instructions in some computer language. Before the program can be used or run, those instructions must be translated into machine language and the machine-language program must be loaded into the computer's main memory. Then, one at a time, the program's instructions are brought into the processor, decoded, and executed. A processor executes the program instructions in sequence until it comes to a "jump" instruction, which causes it to start executing the instructions in another part of the program. A typical instruction brings data into one of the registers, sends data out to the memory or to an output device, or executes some computation on the data in the registers.

### 1.1.2 The Memory

The memory of a computer consists of a very large number of storage locations called **bits**, which is short for "binary digits." Each bit can be turned either off (to store a 0) or on (to store a 1). All

memory in a computer is made out of bits or groups of bits, and all computation is done on groups of bits. The bits in memory are organized into a series of locations, each with an address. In personal computers, each addressable location is eight bits long, called a **byte**. In larger computers and older computers, the smallest addressable unit often is larger than this; in a few machines, it is smaller. Figure 1.2 is a diagram of main memory, depicted as a sequence of boxes with addresses.

A byte could contain data of various types. It could contain one character, such as 'A', or a small number. The range of numbers that can be stored in one byte is from 0 to 255 or from -128 to +127, which is not large enough for most purposes. For this reason, bytes generally are grouped into longer units. Two bytes are long enough to contain any integer between 0 and 65,536, while four bytes can hold an integer as large as 2 billion.<sup>1</sup>

Traditionally, a **word** is the unit of data that can pass across the bus to or from main memory at one time.<sup>2</sup> Small computers usually have two-byte words; workstations and larger computers have words that are four bytes or longer.

A computer might have several different types of **memory** to achieve different balances among capacity, cost, speed, and convenience. The major types are cache memory, main memory, secondary memory, and auxiliary memory. These are diagrammed in Figure 1.3 and discussed next.

**Cache memory.** The fastest and most expensive kind of memory is a **cache**. Some machines have small caches to speed up access to frequently used data that are stored in the main memory. The first time an item is used, it is loaded into the cache. If it is used again soon, it is retrieved from the cache rather than from the main memory, reducing the access time. Cache memories are small because they are very expensive. Their small capacity limits the extent to which they can improve performance.

**Main memory.** The computer's main memory (sometimes called *RAM*, for random access memory) is where the active program (or programs) is kept with its data. Sometimes the operating system, which controls the computer,<sup>3</sup> also is kept in RAM; otherwise it is in read-only memory.

**OS-ROM memory.** **ROM** stands for "read-only memory." An operating system (OS) in ROM is a real convenience for the user for two reasons. First, it is installed in the computer and need not be brought in from a disk. This saves time when the system is turned on. Second, ROM is read-only memory, which means its contents can be read but not changed, either accidentally or on purpose. A partly debugged program sometimes can "run wild" and try to store things in memory locations allocated to some other process that is simultaneously loaded into the computer. With the operating system in ROM, most of the system is protected from this kind of random modification. The only remaining vulnerable part of the system is the area near memory address 0, which is called **low memory** and contains the locations used to communicate between the operating system and the input/output devices.

The disadvantage of using ROM for an operating system is that it is difficult and expensive to improve the system or correct errors in it. When a company wishes to release a new version of a ROM operating system, the code must be recorded on a set of ROM chips. The computer owner must buy a set, remove the old ROM chips, and install the new ones.

**Direct-access memory.** Direct-access memory, also called *secondary memory*, is needed because even the largest main memory cannot store all the information we need. Data files and software packages are kept in secondary memory when not in use.<sup>4</sup>

---

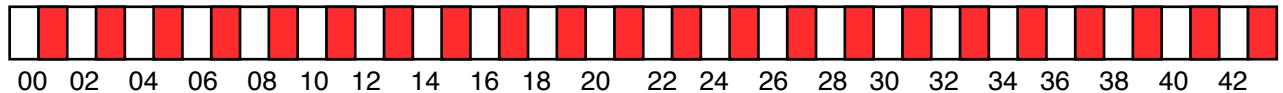
<sup>1</sup>Number representation is covered in Chapters 7 and 15.

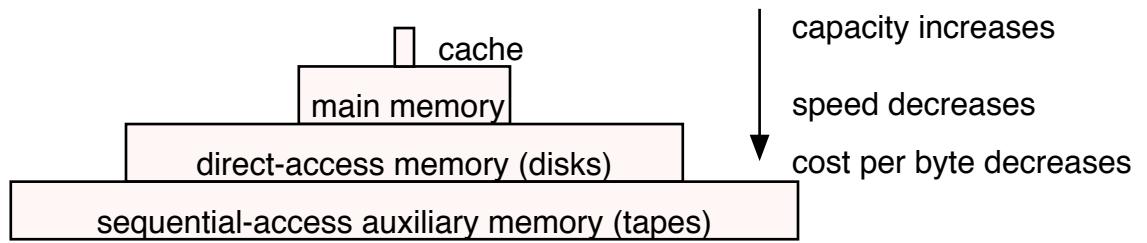
<sup>2</sup>In machines where the bus transports only one byte, *word* means two bytes and *long word* means four bytes.

<sup>3</sup>Section 1.2 covers operating systems.

<sup>4</sup>As memory capacities have expanded, our desire to store information has kept pace, so that this statement is as true today as it was when direct-access memories were the size of today's main memories.

- Memory is a very long series of bytes; we show only the first few here.
- Every byte has an address (only the even addresses are shown here).
- In this picture, even-address bytes are white; odd-address bytes are gray.
- We diagram the addresses outside the boxes because they are part of the hardware. They are *not* stored in memory.
- The addresses in the diagram begin with the address 0 and continue through 43.





A **CD-ROM** (compact disk, read-only memory) is an optical disk storage device, like an audio disc except that it is used to store various types of data, not just music data. A CD-ROM reader contains a laser that reads the minute marks etched into the surface of the disk. Once a CD-ROM has been used to record data, it cannot be reused to record different data. Large collections of data of interest to many people are recorded and distributed on CD-ROMs.

Hard disks and diskettes are the most common direct-access memory devices today. Through the years, the physical size of these devices has decreased steadily, and the amount of information they can hold has greatly increased. As technology has progressed, we have been able to store the bits closer and closer together, enabling us to simultaneously decrease size and increase capacity.

**Auxiliary memory.** Today's disks can store large volumes of information: 2 gigabytes (2 billion bytes) now is a common disk capacity. However, most businesses and individuals find that 2 billion bytes of secondary memory is not enough to meet all of their needs. Larger, cheaper memory devices are needed to store backup (duplicate) copies of the files on the hard disk and infrequently used files. Magnetic tapes and tape drives meet this need for auxiliary storage. A tape has very large capacity but must be read or written sequentially. This makes retrieving a file from a tape that contains hundreds of files or recording a new file on the end of the tape very slow and inconvenient.

### 1.1.3 Input and Output Devices

Input and output devices permit communication between human beings and the CPU. The most common input devices include keyboards, mice, and track balls. Typically, output is displayed either on a video screen or via a printer. Since humans and the computer speak different languages, some translation is needed. Using a mouse allows the human to point or click at a portion of the screen to convey a screen position and an intent to the computer. Interactions with a keyboard, the screen, or a printer take place using the English language and some hardware-level translation.

In a simple view of the computer world, hitting the Z key on a keyboard causes a Z to go into the computer, after which that Z can be sent to the video screen or a printer and reappear as a Z. While this usually is true, it is very far from a direct or necessary connection.

Consider the keyboard. When you type the key in the lower left (usually marked with a Z), the information that goes into the keyboard controller is the coordinates of that key, not a Z. Somewhere a code table says "bottom row, first key means Z." When the same keyboard is used in Germany, the key in that position is marked with a Y, and the code table says "bottom row, first key means Y." Similarly, changing the wheel on a daisy-wheel printer changes the letter that prints on the paper.

Translation codes are arbitrary, and several codes are in common use. The most common character code for personal computers is named ASCII (American Standard Code for Information Interchange), a seven-bit code that supports upper- and lowercase alphabets, numerals, punctuation, and control characters. Each device has its own set of codes, but the codes built into one device are not necessarily the same as the codes built into the next device. Some characters, especially characters like tabs, formfeeds, and carriage returns, are handled differently by different devices and even by different pieces of system software running on the same device. For example, the character whose ASCII code is 12 is named *formfeed* in the ASCII code table. The idea of the code makers was that a printer would eject the paper when it received this code. The program in each printer's controller was supposed to look for this code and handle it, and most did. Today, also, some video controllers are programmed to respond to a formfeed character in an analogous way, by clearing the screen.

As a computer user, you need to be aware that equipment not designed to be used together might be incompatible in unexpected ways. You may find computer systems in which the label on the key that you type, the letter that shows on the screen, and the one that comes out of the printer are all different. This can happen because all three depend on software interpretation as well as hardware capabilities.

### 1.1.4 The Bus

The bus is the pathway between the processor and everything else. It consists of two sets of wires: one set has enough wires to transmit an address; the other set transmits data. When the processor needs a data item, it puts the address of the item in the memory address register (MAR) and issues a **fetch** command. The memory address register and the memory data register (MDR) sit at the end of the bus line (see Figure 1.1).

When a **fetch** command is given, the address goes from the MAR out over the bus's address lines and a copy of the required data comes back over the bus's data lines to the MDR. Similarly, to store information into the memory, the information and the target address are put into these two registers and a **store** command is given. The information goes out over the bus to the given address and replaces whatever used to be there.

**Control of peripherals and interfaces.** The input and output devices are attached to the bus lines. Each device has its own address and handles the information in its own way. Each device, in fact, has a different set of instruction codes that it can handle and a controller to carry out those instructions. A **device controller** is a small processor connected between the bus and the device that is used to control the action of the device. For example, consider a hard disk. A disk has a controller that understands how to get addresses, disk instructions, and information off the bus and how to put information and signals back on the bus. It knows how to carry out and oversee all the disk operations.

Making all the highly varied devices respond to instructions in a uniform way is the job of **device drivers**. A device driver consists of software that knows about the specific quirks and capabilities of a specific device. It translates the uniform system commands into a form that the device can handle prior to putting the commands on the bus lines. A different driver may be needed for every combination of operating system and hardware. For example, a UNIX driver for a SCSI<sup>5</sup> disk would translate the UNIX **read-disk** command to the SCSI format. The user becomes aware that device drivers exist when he or she wants to install a new device and must also install the appropriate driver.

Between a device's driver and its controller is an interface, a doorway between the bus and the device. There are many kinds of interfaces, with varied transmission properties, which can be classified into two general groups: serial and parallel. A **serial interface** transmits and receives bits one at a time. A **parallel interface** can transmit or receive a byte (or more) at a time, in parallel, over several wires. Parallel interfaces commonly are used for printers; serial interfaces for modems, certain printers, mice, and other slow-speed devices. A MIDI (musical instrument digital interface) is a serial interface used to communicate with electronic musical instruments such as synthesizers and keyboards.

### 1.1.5 Networks

Inside the computer, the various hardware components communicate with each other using the internal bus. It now is common practice to have computers communicate with each other to share resources and information. This is made possible through the use of networks, physical wires (often phone lines) along which electrical transmissions can occur. The extent of these networks is varied. A **local area network** typically joins together tens of computers in a lab or throughout a small company. A **global network**, such as the Internet, spans much greater distances and connects hundreds of thousands of machines but is truly just a joining together of the smaller networks via a set of gateway computers. A **gateway** computer is a bridge between a network such as the Internet on one side and a local network on the other side. This computer also often acts as a **firewall**, whose purpose is to keep illegal, unwanted, or dangerous transmissions out of the local environment.

**Sharing resources.** One use of networks is to let several computers share resources such as file systems, printers, and tape drives. The computers in such a network usually are connected in a **server-client**

---

<sup>5</sup>The small computer systems interface (SCSI) is a standard disk interface.

relationship, as illustrated in Figure 1.4. The server possesses the resource that is being shared. The clients, connected via a **hub** or **switched ethernet connection**, share the use of these resources. The user of a client machine may print out documents or access files as if the devices actually were physically connected to the local machine. This can provide the illusion of greater resources than actually exist, as well as present a uniform programming environment, independent of the actual machine used. This kind of sharing is less practical over larger networks due to delays caused by data transmissions through gateway machines.

**Communication.** The other typical use of networks is communication. E-mail has become a popular way to send letters and short notes to friends and business associates. Chat rooms provide the opportunity for more direct, interactive communication. The World Wide Web makes a wealth of information

available to the average user at home that used to be available only in distant libraries. It also provides new commercial opportunities; people now can shop for many items on the Web and are able to buy specialty items or get bargains that were previously unavailable to them.

Networks also have changed the workplace and work habits. Many professionals use network transmissions between home and office or between two office locations to gain access to essential information when they need it, wherever they are. This may include using a laptop computer that is connected to the network via a cellular phone.

**Distributed computing.** Networks also are used to allow the computers to communicate between themselves. The complexity of many of today's problems requires the use of reserve computing power. This can be achieved by synchronizing the efforts of multiple computers, all working in parallel on separate components of a problem. A large distributed system may make use of thousands of computers. Synchronization and routing of information in such systems are major tasks, among the many performed by the operating system's software, as discussed next.

## 1.2 The Operating System

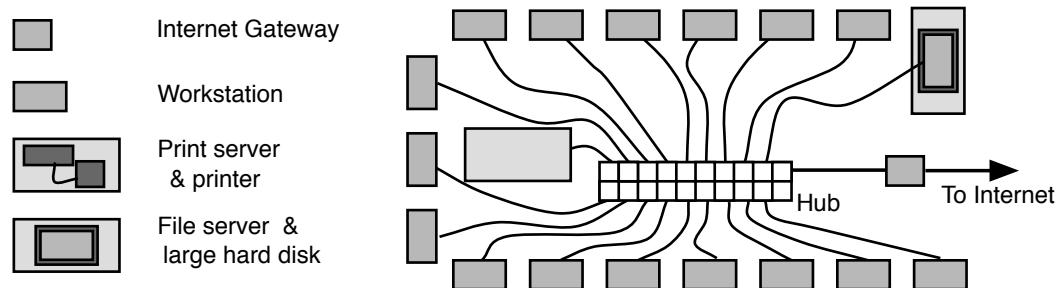
The **operating system** (OS) is the most important piece of system software and the first one you see when you turn on your machine. It is the master control program that enables you to use the hardware and communicate with the rest of the system software. The operating system has several major components, including the **system kernel**, which is the central control component; a **memory management system**, which allocates an area of memory for each program that is running; the **file system manager**, which organizes and controls use of the disks; **device drivers**, which control the hardware devices attached to the computer; and the **system libraries**, which contain all sorts of useful utility programs that can be called by user programs. In addition, a multiprocessing system (described later) has a process scheduler, which keeps track of programs waiting to be run and determines when and how long to run each one.

**Command shells and windows.** Operating systems can be divided into two categories: command-line interpreters and windowing systems. **Command shells**, or command-line interpreters, were invented first and can be used on small, simple machines. A command shell displays a system prompt at the left of the screen and waits for the user to type in a command. Then, the command is executed by calling up some piece of system or user software. When that program terminates, control goes back to the operating system and the system prompt again is displayed.

A newer idea is a **windowing system**. Window-based systems include the Apple Macintosh system, Microsoft Windows (which is an extension of DOS), and NextStep and X-Windows, which provide window interfaces for UNIX. Except for the Macintosh, these windowing systems can run side-by-side with a

---

This is one way that a lab network might be set up. It is not drawn to scale; a hub is a small device that could fit on the corner of a table.



command shell and provide access to it. This is important because command-line interpreters generally provide capabilities that are not available within the window environment.

In a window environment, multiple windows can be displayed on the screen, including perhaps a command window. Windows can be used to display file directories, run programs, and so forth. The user accesses the contents of the windows using a mouse or some other point-and-click device. Seeing your files, moving and copying them, renaming them, and every other thing that you do is much easier in a window environment.

**Multiprogramming systems.** Another way to categorize systems is by whether they can run several programs at the same time or are limited to one at a time. Ordinary personal computers are limited to one process at a time. However, modern personal workstations and large computers, often called *mainframes*, have **multiprogramming** operating systems. **UNIX** is one of the best known and most widely used multiprogramming systems.

Workstations are capable of running a few processes concurrently, and mainframes often can support 50 or 100 users, running the programs in a **time-shared** manner. In time sharing, each user process is given a short slice of CPU time, then it waits while all the other users get their turns. This works because users spend much more time thinking and typing than running their programs. Any request by a program for input or output (I/O) also ends a time slice; the OS initiates the input or output, then selects another process to be run while the I/O happens. The process scheduler is the system component that coordinates and directs all this complex activity.

Each kind of computer must have its own custom-tailored operating system. Some systems are proprietary and have been implemented for only one manufacturer's models. For example, Apple's system for the Macintosh is jealously guarded against copying. Other systems, such as **UNIX** and **DOS**, are widely implemented or imitated. Increasingly, the computer owner has a choice about what system will be installed on his or her hardware.

The choice of hardware is important because it determines what software you can run and what diskettes you can read. Software and file systems are constructed to be compatible with a particular system environment, and they do not work with the wrong system. For example, you cannot read a **UNIX** diskette in a **DOS** system, and a **C compiler** that works under **UNIX** must be modified to work under **DOS**. Windowing systems and multiprogramming are powerful aides to program development. However, both consume large amounts of main memory, disk space, and processing time. Trying to run them with a machine that is not big enough or fast enough is a mistake.

## 1.3 Languages

The purpose of a computer language is to allow a human being to communicate with a computer. Human language and machine language are vastly different because human capabilities and machine capabilities are different. Each kind of computer has its own machine language that reflects the particular capabilities of that machine. Computer languages allow human beings to write instructions in a language that is more appropriate for human capabilities and can be translated into the machine language of many different kinds of machines.

### 1.3.1 Machine Language

Built into the CPU of each computer is a set of instructions that the hardware knows how to execute. The behavior of each instruction and its binary code are documented in the hardware manual. Technically, it is possible to program a computer by making lists of these codes. That is how programming was done 45 years ago.

A machine language program is a sequence of instructions, each of which consists of an instruction code, often followed by one or two register codes or memory addresses. In a machine, these are all

represented in binary.<sup>6</sup> In the early days of computers, when people still wrote programs in machine language, they did not write them in binary because people were (and are) abominably bad at writing long strings of 1's and 0's without making errors. Instead, they used the octal number system,<sup>7</sup> in which information is represented as strings of digits between 0 and 7. Each octal digit translates directly into three binary bits. Thus, a machine language program was written as a long series of lines, where each line was a string of octal digits.

### 1.3.2 Assembly Languages

When people had to use machine language, it took a very long time to write and debug a program. The next development was symbolic assembly language. Instead of writing octal codes, the programmer wrote symbolic codes for instructions and defined a name for each data-storage location. The three lines that follow show how a simple action might look when expressed in assembly language; this code adds two numbers and stores them in a variable named `sum`. The same addition expressed in C would be `sum = n1 + n2;`

```
ldreg *n1, d1    / Load first number into register d1.
add   *n2, d1    / Add second number to the register.
sto   d1, sum     / Store result in the location for sum.
```

A translator, called an **assembler**, analyzed the symbolic codes and assembled machine-code instructions by translating each symbol into its code and assigning memory locations for the data objects used by the program.

Every name and quoted string used in a program must be stored at some address in the computer's memory. To write in machine language, you manually assign an address to each object. Happily, assembly languages and high-level languages such as C free you from concern over these addresses. The programmer declares the names to be used at the top and defines each name by giving it a data type or quoted string value. When the assembler translates this into machine language, it assigns memory locations for these objects.

Assembly languages still are very important for writing programs so closely related to the hardware that high-level languages like C simply have no commands to express them. Many large systems are written primarily in a high-level language but contain some parts coded in assembly language. These portions are part of the system kernel. They work directly with parts of the machine and must operate as efficiently as possible.

### 1.3.3 High-Level Languages

Programming in an assembly language is very tedious. Furthermore, an assembly language is specific to one type of machine and probably very different from assembly languages for machines of other manufacturers. Therefore, assembly language programs are not portable—that is, they are not easily converted for use on other machines. In contrast, programs in languages like FORTRAN and C are highly portable, because compilers for these languages have been created for nearly every kind of computer.

Over the past 40 years many high-level languages have been developed, some of which have stood the test of time and some of which have not. Each instruction in a high-level language translates into several at the assembly or machine language level. Programs written in a high-level language appear much more like English and are more understandable to humans. In this section, we discuss a few widely used programming languages.

---

<sup>6</sup>Binary is the base 2 number system. Information is represented as strings of bits (see Chapter 15).

<sup>7</sup>Octal is base 8.

**The C language.** C is a relatively old<sup>8</sup> language that recently became very popular. It has characteristics of both high-level languages such as Pascal<sup>9</sup> and FORTRAN<sup>10</sup> and low-level languages such as assembly language. You still might see several dialects of C in older programs or books. In 1988, however, the American National Standards Institute (ANSI) adopted a standard for the language, known as ANSI C. This standard was adopted with a few minor changes by the International Standards Organization (ISO) in 1990 and amended in 1994. This new standard is known as ISO C. The phrase *standard C* can be applied correctly to either version of the standard. The changeover to the standard language is nearly complete, so the beginning C student can safely focus all efforts on standard ISO C.

There are several reasons for the recent growth in the use of C:

- As the UNIX operating system has spread, so has C. C is the tool by which much of the power of UNIX is accessed. UNIX has spread because of the very large amount of valuable software that runs under it.
- From the beginning, C was a very powerful language and fun for the experienced programmer to use. However, it lacked certain important kinds of compile-time error checking and therefore was quite difficult for a beginner to use. ISO C incorporates important new error-checking features and has eliminated many hardware dependencies. It developed into a much better language and became suitable for both beginners and experienced programmers.
- C allows large programs to be written in separate modules. This makes it easier to manage large projects, greatly facilitates debugging, and makes it possible to reuse program modules that do common, useful jobs.
- The ISO C library is extensive and standardized. It contains functions for mathematical computation, input and output facilities, and various system utilities.

On the other hand, ISO C remains more error prone than languages of a similar age with similar features, such as Pascal and Ada.<sup>11</sup> It is popular among experienced programmers partly because of the features that cause this error-prone nature:

- C allows the programmer to write terse, compact code that can run very efficiently. Any programmer who is a slow typist appreciates this. Sometimes programmers even make a game of squeezing the unnecessary operations out of their code. On the negative side, compact code can be hard to read and understand unless comments are used liberally to explain it.
- The error-checking system in ISO C is less rigid and more permissive than in competing languages. Expert programmers claim that this permissiveness is an advantage and that the rigid mechanisms in other languages often “get in the way” when they want to do something unusual. However, these rigid systems are easier for the new programmer to understand and to use.
- C supports bit-manipulation operators that can select or change a single bit or group of bits in a number. These are very important in system programs that must interface to hardware devices that set and test values in specific memory locations. However, working with arbitrary machine addresses and bit patterns must be done with extreme care to avoid errors.
- No restrictions are placed on the use of pointers. (A **pointer** is a variable representing the location, as opposed to the value, of data.) This permits the use of some very efficient computational methods, at the potential cost of destroying information anywhere in memory when an error is made in setting a pointer value. Unfortunately, such errors are common, and many result in system crashes and the need to reboot the system.

---

<sup>8</sup>Created originally in 1972, the language has been updated and expanded several times.

<sup>9</sup>Pascal has been used most extensively as an instructional language in universities.

<sup>10</sup>Short for “formula translator,” this language was developed primarily for doing scientific calculations.

<sup>11</sup>Ada is a programming language developed in the 1970s to support large-scale, portable application systems.

**C and FORTRAN.** FORTRAN is a very old language that has been used by engineers and scientists since the infancy of computers. Originally, it was a language for scientific computation, and it still serves that purpose. Over the years, the language has been updated, revised, and expanded, but its primary focus remains high-performance numerical computation. A massive amount of scientific programming now exists in the form of FORTRAN libraries and FORTRAN application programs that are used, and shared, by scientists worldwide. The FORTRAN libraries are extremely efficient, reliable, and trusted.

Because many engineering departments are acquiring UNIX workstations, C is beginning to supplant FORTRAN-77 in many engineering applications. This has some advantages. FORTRAN-77 still bears the burden of being an old language. It is full of unnecessary complications and nonuniform conventions. Much of the space in a FORTRAN textbook is spent explaining how to *write* the language correctly. In contrast, a C textbook has a much simpler language to present and can spend more time explaining how to *use* the language well.

On the other hand, FORTRAN-77 is a “safer” language. A program can get into trouble in very few ways that will cause a system crash or cause the result of a seemingly correct expression to be nonsense. C is prone to these problems, even when the programmer avoids using the advanced parts of the language. When a C programmer begins to use pointers, debugging becomes substantially more difficult than it ever could be in FORTRAN-77. Nonetheless, C is here, and thousands of former FORTRAN programmers are beginning to use it. FORTRAN-to-C conversion programs exist and are being used to make the transition less costly.

**C and C++.** The C++ language extends C to eliminate more causes of error and provide software-engineering tools that are important for large projects. Also, C++ (but not C) is fully compatible with the FORTRAN libraries. This can be a very important consideration for a department switching from FORTRAN to C. C++ is a superset of C. The ordinary line-by-line code in a C++ program is written in C. The extensions involve the way code is organized into modules and the way these modules are used. The C++ extensions are a powerful tool for program organization and error prevention. However, since the entire C language is included as a subset of C++, any error that you can make in C also can be made in C++. The advantages of C++ are there only for those who know how to use them. For beginners, C++ is a more difficult and confusing language than C.

The differences between C and C++ become significant only for moderate to large programs, and only when C++ is used with proper object-oriented design techniques. All software-engineering techniques presented in this book are appropriate for use with both C and C++. The way in which C language elements are presented will lead toward an understanding of the design requirements for C++.

## 1.4 What You Should Remember

### 1.4.1 Major Concepts

- This chapter provides a brief description of computer hardware and software. It describes the parts of the machine a programmer must know to comprehend the operation of a program or buy a personal computer system wisely.
- Computer languages and the process of translation are discussed, and the C language is compared to FORTRAN and C++.

### 1.4.2 Vocabulary

The terms and concepts that follow have been introduced and described briefly. The first and second columns contain terms related to computer hardware and operating systems; the third column relates to the programming process.

CPU	device controller	program
register	device driver	machine language
memory	serial interface	operating system
cache	parallel interface	system kernel
ROM	local area network	system libraries
CD-ROM	global network	command shell
RAM	gateway	windowing system
bit	firewall	multiprogramming
byte	server	assembler
word	client	compiler
clock	memory management system	ANSI C
bus	file system manager	ISO C
hub	floating-point coprocessor	C++

## 1.5 Using Pencil and Paper

### 1.5.1 Self-Test Exercises

1. Which terms on the vocabulary list relate to the computer's processor?
2. Which terms on the vocabulary list relate to the memory of a computer?
3. Which terms on the vocabulary list relate to the peripherals of a computer?
4. Which terms on the vocabulary list relate to a computer network?
5. Which terms on the vocabulary list relate to system software?
6. For what does each of the following abbreviations stand?

a.	ALU	f.	ANSI	k.	LAN
b.	bit	g.	I/O	l.	MIDI
c.	CPU	h.	OS	m.	SCSI
d.	ISO	i.	ROM	n.	MHz
e.	ASCII	j.	WAN	o.	RAM

### 1.5.2 Using Pencil and Paper

1. Choose three terms from *each column* of the vocabulary list in Section 1.4.2. In your own words, give a brief definition for each (a total of nine definitions).
2. What computer will you use for the programming exercises in this course? What kind of processor chip does it have? How big is its main memory? What input and output devices are available for it? Is it attached to a computer network?
3. Have you used a local area network? Why? Have you used the Internet? For what purposes?
4. What operating system runs on the computer that you will use for the programming exercises in this course? Is this a multiprogramming system? What compiler will you use?
5. Explain the difference between
  - (a) a byte and a word.
  - (b) ROM and RAM.
  - (c) cache memory and main memory.
  - (d) a device controller and a device driver.
  - (e) a compiler and an assembler.
  - (f) a command shell and a windowing system.
  - (g) a LAN and a WAN.
  - (h) a gateway and a hub.



## Chapter 2

# Programs and Programming

### 2.1 What Is a Program?

A program is a set of instructions for a computer. Programs receive data, carry out the instructions, and produce useful results. More precisely, the computer **executes a program** by performing its instructions, one at a time, on the supplied data. The instructions a computer is capable of carrying out are very primitive—add two numbers, move a number from here to there in memory, and so forth. Large numbers of instructions are required to carry out even the simplest task, so some programs are thousands or even millions of instructions long.

Nowadays, programs are so big and complicated that they cannot be constructed without the help of the computer itself. A C **compiler** is a computer program whose purpose is to take a desired program coded in a programming language and generate the low-level instructions for the computer from that code. A C compiler is usually used through a program development system called an *integrated development environment*, or *IDE*, that includes an editor and provides windows for viewing code, results, and other relevant information.

The process of submitting a code file to a compiler for translation is called *compilation*. The programmer's code is called **source code** (or C *code* in this case). And the compiled program is called **object code** or **machine code**. One often blurs the distinction between source code and object code by using the word *program* to refer to either. Ideally, the intended meaning will always be clear from the context.

The C *programming language* is the notation used for writing C code. The purpose of this book is to help you learn how to write programs using the C programming language and to use a C compiler and an IDE to generate the instructions that will allow the computer to carry out the actions specified by your program.

**Algorithms.** Many programs are based on algorithms. An **algorithm** is a method for solving a well-structured problem, much as a recipe is a step-by-step process for cooking a particular dish. The method must be completely defined, unambiguous, and effective; that is, it must be capable of being carried out in a mechanical way. An algorithm must terminate; it cannot go on forever.<sup>1</sup> As long as these criteria are met, the algorithm may be specified in English, graphically, or by any other form of communication. Figure 2.1 is an example of a very simple algorithm specified in English.

Thousands of years ago, mathematicians and scientists invented algorithms to solve important problems such as computing areas of polygons and multiplying integers. More recently, algorithms have been developed for solving engineering, mathematical, and scientific problems such as summing a series of numbers, integrating functions, and computing trajectories. A major area of computer science focuses on the creation, analysis, and improvement of algorithms. Computer scientists have invented new algorithms for computing mathematical functions and organizing, sorting, and searching collections of data. The growing power of computer software is due largely to recently invented algorithms that solve problems better and faster.

---

<sup>1</sup>Some programs, generally called *systems programs*, are not based on algorithms and are designed to go on forever. Their purpose is to help operate the computer.

---

To find the average of a set of numbers do the following:

- Count the numbers in the set.
  - Add all of them together to get their sum.
  - Divide the sum by the count.
- 

**Figure 2.1. A simple algorithm: Find an average.**

Algorithms are sometimes explained using a mixture of English and computer language called **pseudocode** (because it looks like computer code, but is not). Figures 2.3, 2.5, and 2.7 show how the simple English statement of the average algorithm from Figure 2.1 progressively is developed into a program specification (Figure 2.3), then into pseudocode (Figure 2.5) and finally into a C program (Figure 2.7).

Algorithms are also sometimes written in C or in a similar high-level language, or as graphical representations (known as **flowcharts**) of the sequence of calculations and decisions that form the algorithm. These forms (rather than pseudocode or English) are normally used when algorithms are published in textbooks or journals.

### 2.1.1 The Simplest Program

A very simple, yet complete, program is shown in Figure 2.2. All it does is print a greeting on the computer's screen. Even though this program is very short, it illustrates several important facts about C programs:

- The first line in this example is called a comment. It is ignored by the C compiler and is written only to provide information for human readers.
- The second line is a preprocessor command. It instructs the C compiler to bring in the declarations of the standard input and output functions so that this program can perform an output operation.
- Every C program must have a function named `main()`. Execution begins at the first statement in `main()` and continues until the `return` statement at the end. The statements are surrounded by a pair of braces.
- This program is a sequence of two statements. The first is a call on the `puts()` function to display a phrase on the computer's screen (`puts()` is the name of one of the standard output functions).
- The statement `return 0;` is always written at the end of a program; it returns control to the computer's operating system after the program's work is done.
- If you enter this program into your own C system, compile it, and run it, you should see the following message on your computer's screen:

Hail and farewell, friend!

---

```
// This simple program is where we begin.
#include <stdio.h>
int main( void )
{
    puts( "Hail and farewell, friend!" );
    return 0;
}
```

---

**Figure 2.2. A simple program: Hail and farewell.**

## 2.2 The stages of program development.

Some programs are short and can be written in a few minutes. However, most programs are larger and more complex than that, and often created by many people working for many months. Whether a program is small or large, the steps in creating it are the same:

1. **Define the problem.** The programmer must know the purpose and requirements of the program he or she is supposed to create.
2. **Design a testing process.** The programmer must figure out how to test the program and how to determine whether or not it is correct.
3. **Design a solution.** The programmer must plan a sequence of steps that starts with entering the input and ends by supplying the information that is required.
4. **Program construction.** After planning comes coding. The programmer codes each step in a computer language such as C, and enters the C code into a computer file using a text editor.
5. **Program translation.** The file of C code is processed by the C compiler. If there are no errors, the result is executable code.
6. **Verification.** The program is run, or executed, on data values from the test plan. The program normally displays results and the programmer must verify that the results are what was expected.

In reality, many of these steps are repeated in a cycle until all errors are eliminated from the program. In the rest of this chapter, each of these steps will be described in more detail.

### 2.2.1 Define the Problem

Most programs are written to solve problems, but you don't start by writing program code or even by planning a solution. Before you can begin to solve a problem, you must understand exactly the nature of the problem. You need to know what data you will start with (if any) and what answers you are supposed to produce.

Suppose a teacher asked a helper to write a small program to compute the average of a student's test scores. The helper might go through these stages in defining the problem:

#### First version of specification.

Goal: Compute a student's exam average.

Input: A list of exam scores.

Output: The average of the scores.

We also must decide how the output will be presented. First, all output should be labeled clearly with a word or phrase that explains its meaning. Also, it is a very good idea to "echo" every input as part of the output. This lets the user confirm that the input was typed and interpreted correctly and makes it clear which answer belongs with each input set. Spacing should be used to make the answers easy to find and easy to read.

**Define constants and formulas.** Some applications require the use of one or more formulas and, possibly, some numeric constants. This information is part of a complete specification, so we add the formula and update the output description:

#### Second version of specification.

Goal: Compute a student's exam average.

Input: A list of exam scores.

Output: Echo the inputs and display the average of the scores.

Formula: average = (sum of all scores) / (number of scores)

---

<b>Goal:</b>	Compute a student's exam average.
Input:	The user will enter exam scores interactively.
Output:	Echo the inputs and print their average.
Constant:	There are 3 exam scores.
Formula:	$Average = (score1 + score2 + score3)/3.0$
Other requirements:	The answer must be accurate to at least one decimal place.

---

**Figure 2.3. Problem specification: Find the average exam score.**

At this stage, the programmer should realize that more information is needed and go back to the teacher to find out how many exams the class had, and whether the program should do anything about strange scores like -1 or 1000. Assume that the teacher's answers are 3 exams, and (for now) don't worry about ridiculous scores. We now have a complete problem description, which is shown in Figure 2.3.

### 2.2.2 Design a Test Plan

Verification is an essential step in program development. When a program is first compiled and run, it generally contains errors due to mistakes in planning, unexpected circumstances, or simple carelessness. Although program testing and verification take place after a program is written and compiled, a **program verification** plan should be created much earlier in the development process, as soon as the program specifications are complete. Designing a test plan at this stage helps clarify the programmer's thoughts. It often uncovers special cases that must be handled and helps create an easily verified design for the solution.

A **test plan** consists of a list of input data sets and the correct program result for each, which often must be computed by hand. This list should test all the normal and abnormal conditions that the program could encounter. To develop a test plan, we start with the problem definition. The first few data sets (if possible) should have answers that are easy to compute in one's head. These test items often can be created by looking at the formulas and constants that are part of the problem definition and choosing values for the variables that make the calculations easy. Figure 2.3 shows a test plan for the exam average problem.

Next, all special cases should be listed. The test designer identifies these by looking at the computational requirements and limitations given in the problem definition. Extreme data values at or just beyond the specified limits should be added to the plan. The next step is to analyze what could go wrong with the program and enter data sets that would be likely to cause failure. For example, some data values might result in a division by 0.

The last data sets should contain data that might be entered during normal use of the program. Since the answers to such problems generally are harder to compute by hand, we list only one or two of them. The results for these items allow the programmer to refine the appearance and readability of the output.

After the coding step is complete, the programmer should return once more to the test plan. If necessary, more test cases should be added to ensure that every line of code in the program is tested at least once.

These guidelines have been stated in general terms so that they are applicable to a wide range of program testing situations. Because of this, the guidelines are somewhat vague and abstract. More examples of test

---

Case	Score1	Score2	Score3	Answer
Easy to compute	10	20	30	20
Special cases	100	100	100	100
	0	0	0	0
Typical cases	66	5	21	30.67
	90	82	89	87

---

**Figure 2.4. Test plan for exam average program.**

1. Declare names for real numbers  $n1$ ,  $n2$ ,  $n3$ , and *average*.
2. Display output titles that identify the program.
3. Instruct the user to enter three numbers.
4. Read  $n1$ ,  $n2$ , and  $n3$ .
5. Add  $n1$ ,  $n2$ , and  $n3$  and divide the sum by 3.0. Store the result in *average*.
6. Display the three numbers we just read so that the user can verify that they were entered and read correctly.
7. Display the *average*.
8. Return to the system.

---

**Figure 2.5. Pseudocode program sketch: Find an average.**

plans are given on the website for this chapter and as part of the case studies in later chapters of this text. These examples will help you gain understanding of verification techniques and learn to build test plans for your own programs.

### 2.2.3 Design a Solution

We are now ready to begin writing the program. We must design a series of steps to go from the data given to the desired result. The steps of most simple programs follow this pattern:

- Display a title that identifies the program.
- Read in the data needed.
- Calculate the answer.
- Print the answer.

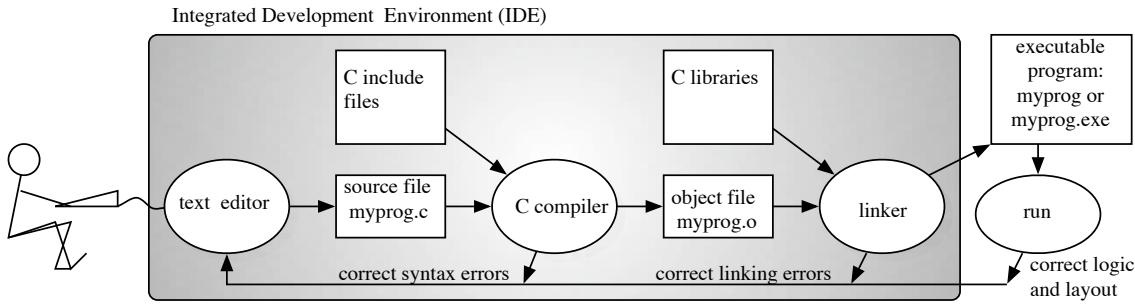
Following this guide and the specification, we write down a series of steps that will solve the problem. Many programmers like to write a *pseudocode* version first, which is a combination of English and programming terms. Figure 2.5 gives a pseudocode version of the exam average program.

## 2.3 The Development Environment

A program **development environment** consists of a set of system programs that enable a programmer to create, translate, and maintain programs. Many modern commercially available compilers are part of an *integrated development environment*, which includes an editor, compiler, linker, and run-time support system including a symbolic debugger. These system programs are used through a menu-driven shell. The stages of program creation are illustrated in Figure 2.6 and described next.

### 2.3.1 The Text Editor

The **text editor** is a tool used to enter a program into a computer file. Text editors permit a typist to enter lines of code or data into a file and make changes and rearrange parts easily. They lack the font and style commands and control over page layout included in a word processor because these “bells and whistles” are neither useful nor desirable in a program file. If you were to use a word processor and enter your program into a normal word-processor document file, the compiler would be unable to translate the program because of the embedded formatting commands.



**Figure 2.6. The stages of program translation.**

When you enter a text editor to create a new program or data file, its screen will show a blank document. To type a program, just start at the top and type the lines of code. This is your source code. The first time you save the program file, you must give it a name that ends in *.c*. Be sure to save your file periodically and maintain a backup copy. When the program is complete, save it again. Data files are created similarly, except that they are given names ending in *.in* or *.dat*.

Text editors come in three general varieties. The very old editors, like the UNIX *ed* editor, are line editors. They operate on one line of a file at a time, by line number, and are slow and awkward to use. Somewhat newer are the full-screen editors, such as UNIX *vi* and older versions of *emacs*, which allow the programmer to move a cursor around the video screen and delete, insert, or change the text under the cursor. They also can delete, copy, or move large blocks of code. Some make periodic, automatic backup copies of your file, which is a tremendous help when the power unexpectedly goes off. To use one of these editors, the programmer enters a command into the operating system interpreter that contains the name of the program to be executed, the names of the files to be used, and sometimes other information regarding software options.

Modern text editors (including the new versions of *emacs*) permit all the necessary text editing operations to be done using a mouse and a text window with scroll bars. They often display the text in color, using different colors for C's reserved words, user-defined words, and comments.

Use the most modern text editor you can find. Scroll bars and mice make a huge difference in the ease of use. A good editor encourages you to write good programs. You will be much more willing to make needed changes and reorganize program parts if it is not much work to do so.

### 2.3.2 The Translator

We use programming languages to communicate with computers. However, computers cannot understand our programming languages directly; they must be translated first. When we write source code in a symbolic computer language, a translator is used to analyze the code, follow the commands, find the declarations, assign memory addresses for the data objects, and determine the structure and meaning of each statement.

There are two kinds of translators: compilers and interpreters. After analyzing the source code, a compiler generates machine instructions that will carry out the meaning of the program at a later time. At **compile time**, it plans what memory locations will be needed and creates an object file that contains machine-language instructions to allocate this memory and carry out the actions of the program. Later, the object code is linked with the library, loaded into memory, and run. At **run time**, memory is allocated and information is stored in it. Then, the machine instructions are executed using these data.

An interpreter performs the translation and execution as a single step. After determining the meaning of each command, the interpreter carries it out. Some languages, such as BASIC and Java, normally are interpreted. Others, such as FORTRAN and C, normally are compiled. In general, compiled code is more efficient than interpreted code.

### 2.3.3 Linkers

An object file is not executable code. Before you can run your program, the linker (another system program) must be called to link it with the precompiled code in the C system libraries. Although the two processes *can* be done separately, linking is normally done automatically when compilation is successful. The result of linking is a **load module** or **executable file**; that is, a file that is ready to load into the computer's memory and execute. The programmer should give this file, which will be used to run the program, a convenient and meaningful name. In some systems, the name automatically will be the same as the name of the source file except that it ends in *.exe*. In UNIX, however, the name of the load module is *a.out* unless the user provides a better name as part of the compile command.

## 2.4 Source Code Construction

You have specified the whole project and designed an algorithm to do the computation. Now you are ready to write the actual code for your program. In the old days, when computers were scarce and expensive, programmers wrote out their code, in detail, on paper so that everything was complete before approaching the computer. This ensured a minimum amount of computer time was consumed. Today, computer time is no longer scarce, so experienced programmers often skip the paper-copy phase; they approach the computer with a detailed sketch of the code but without actually writing it in longhand. Others, especially beginners, still prefer to write the entire program on paper before sitting down at the computer. This permits them to separate the process of writing the code from the difficulties of dealing with a computer and a text editor. It also is helpful in getting "the big picture"; that is, seeing how all the parts of the program fit together. However, whether you are a beginner or an expert, it is important to have at least a well-developed and well-specified program sketch when you begin entering code into the computer. Programming without one is difficult and failure prone, because without a good plan, it is hard to know what to do and what order to do it in.

The steps in code construction are:

1. Create a project and a source file in a proper subdirectory.
2. Write or copy C code into the source file.
3. Build (compile, then link) the C code.
4. Correct compilation and linker errors and rebuild.
5. Run and test the executable program.

This text presents two methods for coding a program: Start from scratch or adapt an existing program to meet the new specification. The first several chapters rely primarily on adapting the programs given as examples in the text. In Chapter 9, after the fundamental concepts of programming have been covered, we discuss the process of building a program from scratch, using a specification.

Here, we continue with the exam average problem. An algorithm was given in Figure 2.1; a specification is given in Figure 2.3 a test plan in Figure 2.4 and a pseudocode program sketch in Figure 2.5. Now we are ready to write C code.

### 2.4.1 Create a Source File

To begin, you should create a new subdirectory (folder) on your disk for use only with this program. By the time a program is finished, it will have several component files (source code, backup, error file, object code, executable version, data, and output). Your life will be much simpler if you keep all the parts of a program together and separate them from all other programs.

Once you have made a directory for the new project, you are ready to create a new source code file in that directory. This can be done by copying and renaming an existing program you intend to modify or by starting with a blank document and storing it under your new program name. Then either type in your

---

```
#include <stdio.h>
int main( void )
{
    double n1, n2, n3;           // The three input numbers.
    double average;              // The average of the three numbers.

    puts( "Compute the average of 3 numbers" );
    puts( "-----" );
    printf( "Please input 3 numbers: " );
    scanf( "%lg%lg%lg", &n1, &n2, &n3 );    // Read the numbers.

    average = (n1 + n2 + n3) / 3.0;        // Average is sum / 3.
    printf( "The average of %g, %g, and %g = %g \n\n",
           n1, n2, n3, average );          // Print answers.
    return 0;
}
```

Sample output:

```
Compute the average of 3 numbers
-----
Please input 3 numbers: 21 5 66
The average of 21, 5, and 66 = 30.6667
```

---

**Figure 2.7. A C program.**

new source code or modify the existing code. When you are finished, print out a copy of your file and look carefully at what you have done. (If your printer is inconvenient or produces output of poor quality, you may find it easier to examine the code on the video screen.) Mark any errors you find and use the text editor to correct them.

### 2.4.2 Coding.

With a firm idea of what the program must do, we can proceed to writing the steps in the C language. Figure 2.7 gives the C code that corresponds to the pseudocode in Figure 2.5. The notes, below, give a quick guide to the meaning of the C statements. Do not try to understand the C code at this time; all will be explained more fully in Chapter 3. This goal in this chapter is only to understand the process of program development, coding, translation, and execution.

1. Like all programs, this one starts with a command to include the declarations from the C input/output library.
2. Next is the line that declares the beginning of the main program. That is followed by a pair of curly braces that enclose the body of the program (declarations and instructions).
3. The first two lines of the program body are declarations that create and name space to store four numbers, *n1*, *n2*, *n3*, and *average*.
4. The words enclosed in /\* ... \*/ on the right end of these lines are comments. They are not part of the C code, but are written to help the reader understand the code.
5. The next two lines print the program title and a line of dashes to make the title look nice. Compare the code to the output, at the bottom of the Figure.
6. The **printf()** statement Instructs the user to enter three numbers at the keyboard. This kind of instruction is called a *user prompt* or, simply, a *prompt*.
7. The **scanf()** statement reads three numbers from the keyboard and stores them in *n1*, *n2*, and *n3*. This ends the input phase of the program, and we leave a blank line in the code.

8. Calculations in C are written much as they are in mathematics. The next line calculates the average and stores it for future use.
9. The next two lines are a `printf()` statement that displays the three input numbers and the average. The first line of this statement has a quoted string called a *format* that shows how we want the answers to be laid out. Each `%g` marks a spot where a number should be inserted. The second line lists the numbers that we want to insert in those spots.
10. The last line of code returns control to the system.

### 2.4.3 Translate the Program and Correct Errors

Before you can run your program, it must be compiled (translated into machine language) and linked with the previously compiled library code modules.

**Compiling.** When we use the compiler, we tell it the name of a C source file to translate. It runs and produces various kinds of feedback. If the job was done perfectly, the compiler will display some indication of success and create a new object file, which contains the machine code instructions that correspond to the C code in the source file. However, it is rare that all the planning, design, coding, and typing are correct on the first try. Human beings are prone to making mistakes and compilers are completely intolerant of errors, even very small ones. Spelling, grammar, and punctuation must be perfect or the program cannot be translated. The result is that the compiler will stop in the middle of a translation with an error comment (or a list of them). The programmer must locate the error, fix it, and use the compiler again.

**Correcting compile-time errors.** When we first entered the average program, we made typographical errors in two lines near the end. The code typed in looked like this:

```
average = (n1 + n2 + n3) / 3.0
print( "The average of %g, %g, and %g = %g \n\n",
       n1, n2, n3, average );
```

The compiler responded with these error comments:

```
Compiling mean.c (2 errors)
mean.c:18: illegal statement, missing ';' after '3.0'
mean.c:19: parse error before "print"
```

The problem is very clear: the line that computes the average needs a semicolon. This was corrected and the code was recompiled.

**Interpreting error comments.** Error comments differ from system to system. Language designers try to make these comments clear and helpful. However, especially with C, identifying the source of the error is sometimes guesswork, and the comments are often obscure or even misleading. For example, the comment `undefined symbol` might indicate a misspelling of a symbol that is defined, and `illegal symbol line 90` might mean that the semicolon at the end of line 89 is missing. Worse yet, a missing punctuation mark might not be detected for many lines, and an extra one might not be detected at all—the program just will not work right.

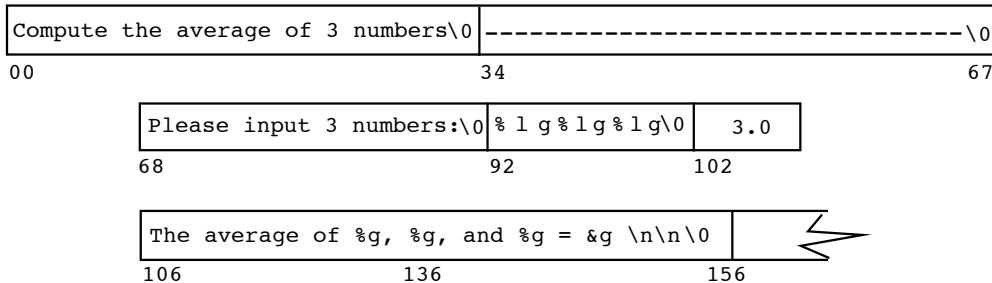
If there is a list of error comments, do not try to remember them all. Transfer the error comments to Notepad or to a new file and either print it or use a part of your screen to display it<sup>2</sup>. Then fix the first several problems. Very often, one small error near the beginning of a program can cause the compiler to become confused and produce dozens of false error comments. After fixing a small number of errors, recompile and get a fresh error listing. Often, many of the errors will just go away.

It takes some practice to be able to interpret the meaning of error comments. Sometimes beginners interpret them literally and change the wrong thing. When this happens, a nearly correct program can get

---

<sup>2</sup>In some systems, using the print-screen button is convenient

This diagram shows how the constant values used in the program of Figure 2.7 might be stored in main memory. Later chapters will explain why.



**Figure 2.8. Memory for constants.**

worse and worse until it becomes really hard to fix. Anyone who is not sure of the meaning of an error comment should seek advice from a more experienced person.

**Correcting linking errors.** The linking process also may fail and generate an error comment. When a **linking error** happens, it usually means that the name of something in the library has been misspelled or that the linker cannot find the system libraries. The programmer should check the calls on the specified library function for correctness. If the compiler is newly installed, a knowledgeable person should check that it was installed correctly. When we tried to link our program, the linker produced this error comment:

After fixing the semicolon error in our program, the compiler finished successfully, but the linker did not:

```
Compiling mean.c (1 warning)
mean.c:19: warning: implicit declaration of function 'print'
Linking /mean/build/mean (1 error, 1 warning)
ld: warning prebinding disabled because of undefined symbols
ld: Undefined symbols:
    _print
```

The linker could not find a function named `print()` in the standard library. I changed it to `printf()`. This time, both compilation and linking were successful and we were ready to run the program.

## 2.5 Program Execution and Testing

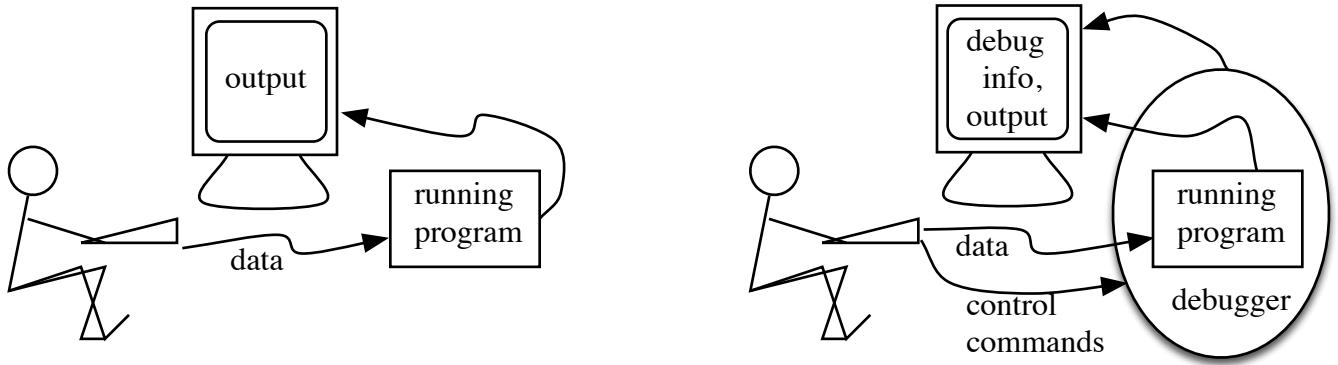
### 2.5.1 Execute the Program

When a program has been successfully compiled and linked, the testing phase begins. An integrated development environment usually provides a icon or a RUN button to run the program; if so, just click on it. If not, you can start execution by clicking the mouse on the program's icon or typing the program's name on a command line. The operating system will respond by loading the executable program into the computer's memory. The memory will then contain the program instructions, any constants or quoted strings in the program, and space for the program's data. Figure 2.8 shows how the constants and data might be laid out in the memory for the program in Figure 2.7.

When loading is complete, the operating system transfers control to the first line of the program. At this time, you should begin to see the output produced by your program. If it needs input, type the input and hit the Enter key. When your program is finished, control will return to the system. When the program's instructions are executed, the computer's devices perform input and output and its logic circuits perform calculations. These actions produce data and results that are stored in the memory. In an integrated development environment, the system screen may disappear while your program is running and appear again when it is finished.

---

A program can be tested by running it directly (left) or through an on-line debugger (right).




---

**Figure 2.9.** Testing a program.

When a program is running, the contents of the memory cells change every time a `read` or `store` instruction is executed. In some systems, a program can be run with a debugger. A **debugger** is another program that runs the program for you. As shown in Figure 2.9, you communicate with the debugger, and it interprets your program instructions one at a time, step by step. You can ask to see the results of computations and the contents of selected parts of memory after each step and, thus, monitor how the process is proceeding. Figure 2.10 shows, step by step, how the values stored in the memory change when the program in Figure 2.7 is executed.

A debugger can be a powerful tool for figuring out why and how an error occurs. Even though the C language is fully standardized, C debuggers are not. Each one is different. Because of this, it is helpful to learn other ways to monitor the progress of a program. Such techniques are presented in the next section.

---

This diagram shows how the values stored in main memory change when we run the program from Figure 2.7. A ? indicates that any garbage value might exist at that location.

When the program is ready to run, the memory area for variables might look like this:

	average	n3	n2	n1					
→	964	968	972	976	980	984	988	992	996

After returning from `scanf()`, with inputs of 21, 5, and 66,

	average	n3	n2	n1					
→	964	968	972	976	980	984	988	992	996

After computing the formula and storing the answer,

	average	n3	n2	n1					
→	964	968	972	976	980	984	988	992	996

---

**Figure 2.10.** Memory during program execution.

### 2.5.2 Test and Verify

Many errors are caught by the compiler, some by the linker. When a program finally does compile and link without warnings, though, the process of finding the **logical errors** begins. The primary method for detecting logical errors is to test the program on varied input data and look carefully at the answers it prints. They might or might not be correct. Sometimes a program operates correctly on some data sets and incorrectly on others. Sometimes an unusual data condition will cause a program to crash or begin running forever. In any case, *you must verify the correctness of your answers.*

You now are ready to use your testing plan. Run your program and enter each of the data sets in your plan, verifying each time that the computer got the same answer you got by hand calculation. Compare the correct answer on your test plan to the answer the computer has printed. Are they the same? We tested our program using the plan in Figure 2.4; the output from our fourth test is shown here:

```
Compute the average of 3 numbers
-----
Please input 3 numbers: 21 5 66
The average of 21, 5, and 66 = 30.6667
```

This answer (and the others) matched the expected answers, so we have some confidence that the program is correct. If the answer is not correct, why not? Where is the error—in the test plan or in the program code? If the test plan has the correct answer, you must analyze the code and find the reason for the error; that is, you must find the logical error, better known as a *program bug*. The useful debugging method of using printouts is described next. Another technique (parse trees) is given in Section 4.8.

**Debugging: locating and correcting errors.** As you start writing your own programs, you must learn how to find and eliminate programming errors. Debugging printouts are a powerful technique for locating a logical error in a program. When your program computes nonsense answers, add debugging printouts until you discover the first point at which the intermediate results are wrong. This shows you where the problem is. With that information (and possibly some expert help), you can deduce why the code is wrong and how to fix it. In an integrated development environment, the debugger can be used to provide similar information.

Beginners tend to form wrong theories about the causes of errors and change things that were right. Sometimes this process continues for hours until a program that originally was close to correct becomes a random mess of damaging patches. The best defense against this is to be sure you understand the reason for the error before you change anything. If you do not understand it, ask for help. Be sure to save a backup copy of your program when you modify it. This can be helpful if you need to undo a “correction.”

**Finishing the test.** Whether you find an error or a needed improvement, the next step is to figure out how to fix it and start the edit–compile–link–test process all over again. Eventually the output for your first test case will be correct and acceptably readable. Then the whole testing cycle must begin again with the other items on the test plan: Enter a data set, inspect the results critically, and fix any problems that become evident. When the last test item produces correct, readable answers, you are done.

**Polishing the presentation.** Now that we are sure the answers are correct, we look at the output with a critical eye. Is it clear and easy to read? Should it be improved? Is everything spelled correctly? Does the spacing make sense? Small changes sometimes make big improvements in readability.

## 2.6 What You Should Remember

### 2.6.1 Major Concepts

Facts about programs.

1. **Translation.** Programs are written in a computer language and converted to machine code by a translator, called a *compiler*.
2. **Syntax.** Programs must conform exactly to the rules for spelling, grammar, and punctuation prescribed by the language. Not every piece of source code describes a valid program. To be valid, every word used in the source code must be defined and the words must be arranged according to strict rules.
3. **Errors.** An invalid program can contain three kinds of errors: compilation errors, linking errors, and run-time errors. Invalid program code results in **compilation errors** or **linking errors** when translated. A runtime error happens when the machine malfunctions or the program tries to do an illegal operation, such as dividing by zero.
4. **Semantics.** A program, as a whole, has a *meaning*. The meaning of a C program is its effect when it is translated and executed.<sup>3</sup> Ideally, the meaning of a program is what the programmer intended. If not, we say that the program contains a bug, or **logic error**.

### The stages of program development.

1. Problem definition.
2. Design of a testing process.
3. Design of a solution.
4. Program construction.
5. Program translation.
6. Verification.

### 2.6.2 The moral of this story.

If you wish to avoid grief in your programming, you should take to heart these proverbs:

1. It is harder than it seems it ought to be.
2. The program itself is like the tip of an iceberg: It rests on top of a lot of invisible work and it will not float without that work.
3. Attention to detail pays off. Compilers expect perfection.
4. Debugging seldom is an easy process, and if a program is not well-structured, it can become a nightmare.
5. Every development step skipped is hours wasted. (Beginners rarely understand this.)
6. You cannot start a program the day before it is due and finish it on time.

### 2.6.3 Vocabulary

These are the most important terms and concepts presented or discussed in this chapter.

program	development environment	compile time
algorithm	text editor	compilation error
pseudocode	compiler	linking error
test plan	linker	execute a program
source code	debugger	run time
object code	debugging printouts	run-time error
executable file	statement	logical error
load module	declaration	program verification

---

<sup>3</sup>In some ways, the meaning of a C program depends on the hardware on which it is executed. Although it always will produce the same results when executed on the same machine with the same data, the results might be different when executed on a different kind of computer.

## 2.7 Exercises

### 2.7.1 Self-Test Exercises

1. Use a dictionary to find the meaning and pronunciation of the word “pseudo”. Use that definition to explain the meaning of “pseudocode”.
2. Explain why a programmer must use a text editor, not a word processor.
3. Explain why a beginner usually fails to finish a program on time when he or she starts it the day before it is due.
4. True or false: A program is ready to turn in when it compiles, runs, and produces results. Please explain your answer.
5. True or false: When a program finishes running, it returns control to the operating system. Please explain your answer.
6. Explain the difference between
  - (a) an algorithm and a program.
  - (b) a command and a declaration.
  - (c) a compiler and an interpreter.
  - (d) source code and object code
  - (e) an object code file and a load module.

### 2.7.2 Using Pencil and Paper

1. A test plan is a list of data inputs and corresponding outputs that can be used to test a program. This list should start with input values that are easy to check in your head, then include input values that are special cases and values that might cause trouble, perhaps by being too big or too small.
  - (a) Following the example in Figure 2.3, create a problem specification for a program to convert Fahrenheit temperatures to Celsius using the formula

$$\text{Celsius} = (\text{Fahrenheit} - 32.0) * \frac{5.0}{9.0}$$

Specify the following aspects: scope, input or inputs required, output required, formulas, constants, computational requirements, and limits (if any are necessary) on the range of legal inputs.

- (b) Write a test plan for this algorithm. Include inputs that can be checked in your head, those that will test any limits or special cases for this problem, and typical input values. For each input, list the correct output.
  - (c) Using English or pseudocode, write an algorithm for this problem. Attempt to verify the correctness of your algorithm using the test plan. (Do not attempt to actually write the program in C.)
2. Explain the difference between
  - (a) pseudocode and source code.
  - (b) a text editor and a word processor.
  - (c) compile time and run time.
  - (d) program verification and program execution.
3. Given the short C program that follows,
  - (a) Make a list of the memory variables in this program.
  - (b) Which lines of code contain operations that change the contents of memory? What are those operations?
  - (c) Which lines of code cause things to be displayed on the computer’s screen?

```

int main( void )
{
    double square;
    double number;

    printf( "Enter a number: " );
    scanf( "%lg", &number );
    square = number * number;
    printf( "The square of %g is %g\n", number, square );
    return 0;
}

```

### 2.7.3 Using the Computer

1. Decide which computer system you will use for these exercises. If it is a shared system, find out how to create an account for yourself.

- (a) Create an account and a personal directory for your work.
- (b) Find out how to create a subdirectory on your system. Create one called `info`.
- (c) You will use a text editor to type in your programs and data files. Some C systems have a built-in text editor; others do not. Find out what text editor you will be using and how to access it. Create a text file (not a program) containing your name, address, and telephone number on separate lines. Next, write the brand of computer you are using and the name of the text editor. Then write a paragraph that describes your past experience with computers. Save this file in your `info` directory and email it to your teacher.
- (d) Find out how to print a file on your system. Print out and turn in the file you created in (c).

2. Given the short C program that follows,

- (a) Make a list of the memory variables in this program.
- (b) Which lines of code contain operations that change the contents of memory? What are those operations?

```

int main( void )
{
    double base;          /* Input variable. */
    double height;        /* Input variable. */
    double area;          /* To be calculated. */

    printf( "Enter base and height of triangle: " );
    scanf( "%lg", &base );
    scanf( "%lg", &height );
    area = base * height / 2.0;
    printf( "The area of the triangle is %g\n", area );
    return 0;
}

```

3. Write a simple program that will output your name, phone number, E-mail address, and academic major on separate lines.



# Chapter 3

## The Core of C

The C language is a powerful language that has been used to implement some of the world's most complex programs. However, by learning a modest number of fundamental concepts, a newcomer to C can write simple programs that do useful things. As with any language, the beginner needs a considerable amount of basic information before it is possible to read or write anything meaningful. In this chapter, we introduce some of the capabilities of C by using them in simple, but practical, programs and explaining how those programs work. These examples introduce all the concepts necessary to read and write simple programs, providing an overview of the central parts of the language. Later chapters will return to all of the topics introduced here and explain them in greater detail.

### 3.1 The Process of Compilation

In Chapter 2, we discussed the process of creating a C program. One of the stages in that process is compilation (Section 2.2). The compilation step can also be broken into stages, as shown in Figure 3.1. It helps a programmer to understand what these stages are, and the problems that can arise at each stage.

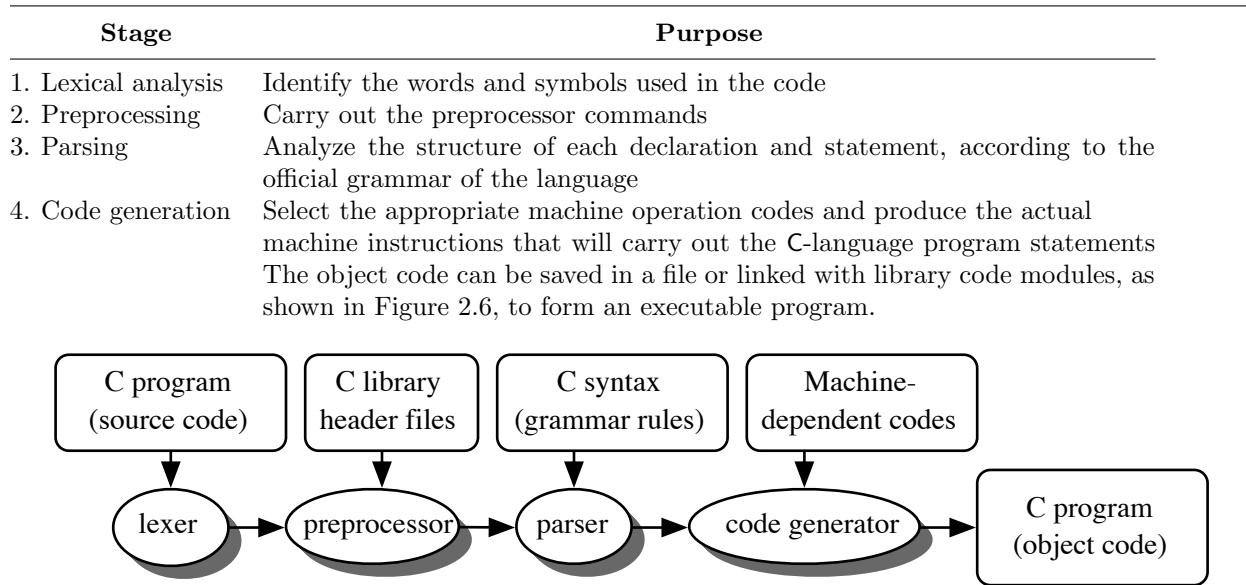


Figure 3.1. The stages of compilation.

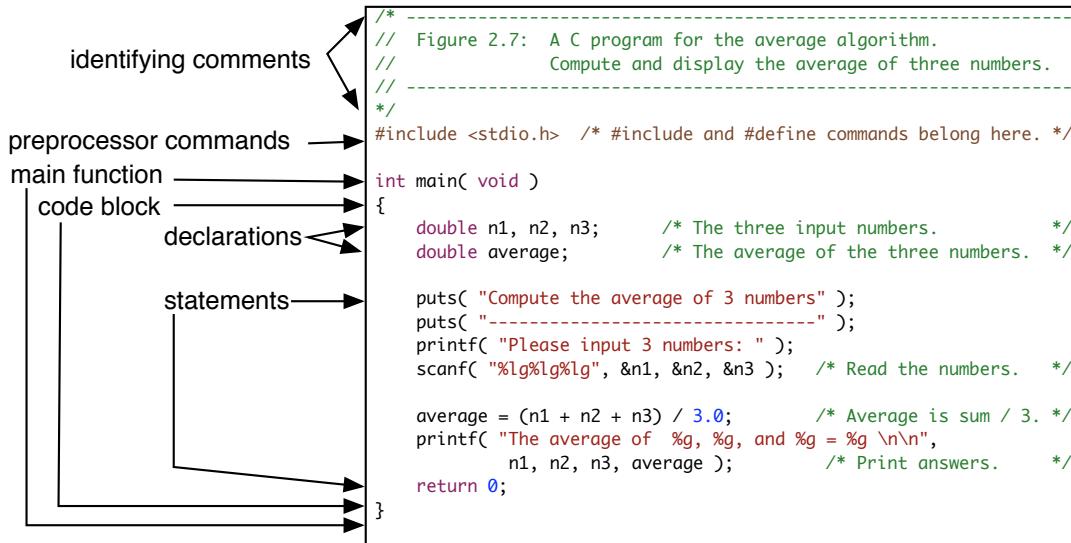


Figure 3.2. How to lay out a simple program.

**Lexical analysis:** Identify the words and symbols used in the code. Some words are built into C and form the core of the language; these are called *keywords* and are listed in Appendix C. New words can be defined by the programmer by giving a declaration. For example, in Figure 2.7, two declarations were used to define the words `n1`, `n2`, `n3`, and `average`.

**Preprocessing:** Carry out the preprocessor commands, which all start with the `#` symbol on the left end of the line. These commands are used to bring sets of C library declarations into your program. In this chapter, you will see that they can also be used to define symbolic names for constants.

**Parsing:** Analyze the structure of each declaration and statement according to the official grammar of the language. When the compiler parses your code it may discover missing punctuation, incomplete statements, bad arithmetic expressions, and various other kinds of structural errors.

**Code generation:** Select the appropriate machine operation codes and produce the actual machine instructions that will carry out the C-language program statements. If the compiler can parse your code correctly, and can find definitions for all of your symbols, it will generate machine code and store it in an object-code file.

## 3.2 The Parts of a Program

### 3.2.1 Terminology.

A C program is a series of comments, preprocessor commands, declarations, and function definitions. The function definitions contain further comments, statements, and possibly more declarations. Each of these program units is composed of a series of words and symbols. We define these terms very broadly in the next several paragraphs; their meaning gradually should become clearer as you look at the first several sample programs. As you read these definitions, you should refer to the diagram in Figure 3.2

**Comments.** At the top of a program, there should be a few lines that start with `//` or are enclosed between the symbols `/*` and `*/`. These lines are comments that supply information about the program's purpose and its author. Their purpose is to inform the human reader, not the computer, and so they are ignored by the

C Category	In English	Purpose
Identifiers	Nouns	Used to name objects
Data types	Adjectives	Used to describe the properties of an object
Operators	Verbs	Denote simple actions like add or multiply
Function calls	Verbs	Denote complex actions like finding a square root
Symbols	Punctuation	Symbols like a semicolon or # are used to mark the beginning or the end of a program unit
Symbols	Grouping	Pairs of parentheses, braces, quotes, and the like are used to enclose a meaningful unit of code

Figure 3.3. Words in C.

compiler. Comments also can and should appear throughout the program, wherever they could help a reader understand the form or function of the code.

**Preprocessor commands.** Also, ly a series of commands at the top of a program tell the C compiler how to process the pieces of a program and where to look for essential definitions. These **preprocessor commands** all start with the symbol # and are handled by the C preprocessor before the compiler starts to translate the code itself, as indicated in Figure 3.1.

**Words.** Many of the words used to write a program are defined by the C language standard; these are known as *keywords*. A list of keywords can be found in Appendix C. Other words are defined by the programmer. These words can be grouped into categories analogous to English parts of speech; the table in Figure 3.3 lists the kinds of words in C and their English analogs.

**Declarations.** The purpose of a **declaration** is to introduce a new word, or **identifier**, into the program’s vocabulary. It is like a declarative sentence: it gives information about the objects a program will use. In C, a declaration must be written in the program prior to any statement that uses the word it defines.

**Statements.** A **statement** is like an imperative sentence; it expresses a complete thought and tells the compiler what to tell the computer to do. Just as an English sentence has a verb and a subject, a typical C statement contains one or more words that denote actions and one or several names of objects to use while doing these actions. The objects we study in this chapter are called *variables* and *constants*; the action words are assignment statements, arithmetic operations, and function calls. An entire program is like an essay, covering a topic from beginning to end.

**Blocks.** Sometimes, C statements are grouped together by enclosing them in braces. Such a group is called a **block**. Blocks are used with **control statements** to into action units that resemble paragraphs.

**Putting it all together.** This chapter shows how to use a subset of the C language we call the *beginner’s toolbox*. It consists of the basic elements from each category: comments, preprocessor commands, declarations, objects, actions, and control statements. Each section will focus on one or a few elements and use them in a complete program. The accompanying program notes should draw your attention to the practical aspects of using each element in the given context.

The sample programs should help you gain a general familiarity with programming terminology and the C language and provide examples to guide the first few programming efforts. Each topic is revisited in more depth in later chapters. As you read this material, try to understand just the general purpose and form of each part. Then test your understanding by completing one of the skeletal programs given at the end of the chapter.

### 3.2.2 The main() program.

Every C program must have a function named `main()`, with a first line like the one shown here, followed by a block of code in braces. Identifying comments and preprocessor commands come before `main()`. The last line of code in `main()` is `return 0`. This much never changes.

The keyword `void` appears in the parentheses after `main()`. This means that `main()` does not receive any information from the operating system. (Some complex programs do receive data that way.) The type name

`int` appears before the name `main()` to declare that `main()` will return a termination code to the operating system when the program finishes its work.

The `{` and `}` (curly bracket) symbols may be read as “begin” and “end.” The braces and everything between them is called a **program block**. For simplicity, we usually use the shorter term **block**. Every function has a block that defines its actions, and the statements in this block are different for every program. They are carried out, in the order they are written, whenever the function is executed. A simple sequential program format is described below; it has three main phases: input, calculation and output. Most short programs follow this pattern.

```
int main( void )
{
    Variable declarations, each with a comment that describes its purpose.
    An output statement that identifies the program.
    Prompts and statements that read the input data.
    Statements that perform calculations and store results.
    Statements that echo input and display results for user.
    return 0;
}
```

### 3.3 An Overview of Variables, Constants, and Expressions

Before we can write a program that does anything useful, we need to know how to get information into and out of a program and how to store and refer to that information within the program. This section focuses on how a program can define and name objects and how those objects can be used in computations. Several ways are introduced to write variable declarations and constant definitions. Formal rules and informal guidelines for naming these objects are discussed and informal rules for diagramming objects are presented.

#### 3.3.1 Values and Storage Objects.

A **data value** is one piece of information. Data values come in several types; the most basic of these are numbers, letters, and strings of letters. When a program is running in a computer, it reads and writes data values, computes them, and sends them through its circuits. A computed data value might stay for a while in a CPU register, be stored into a variable in memory, or be sent to an output device. C keeps track of where data values are when they are being moved around within the computer, but you must decide when to output or store a value (assign it to a variable). C also permits you to give a symbolic name to a value.

We use the term *storage object*, or simply *object* to refer to any area of memory in which a value can be stored<sup>1</sup>. Objects are created and named by declarations. Each one has a definite type and an address. Many, but not all, contain values (the rest contain garbage). Some are variable, some are constant.

#### 3.3.2 Variables

A **variable** is an area of computer memory that has been given a name by the program and can be used to store, or remember, a value. You can visualize it as a box that can hold one data value at a time.

The programmer may choose any name (there are certain rules to follow when picking a name) for a variable, so long as it is not a keyword<sup>2</sup> in the language. Good programmers try to choose meaningful names that are not too long.

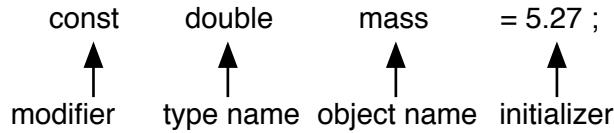
Each variable can contain a specific **type** of value, such as a letter or a number. In this chapter, we introduce the first three<sup>3</sup> data types:

- `char` is used to store characters (letters, punctuation, digits)
- `double` is used for real numbers such as 3.14 and .0056. This types can represent an immense range of numbers but does so with limited precision.
- `int` is used for integers such as 31 and 2006.

<sup>1</sup>This terminology is normal in C, and is consistent with the use of the term in C++. However, the term “object” in Java is used only for instances of a class.

<sup>2</sup>Keywords are words such as `main()` and `void` that have preset meanings to the translator and are reserved for specific uses. These are listed in Appendix C.

<sup>3</sup>Other types are introduced in Chapters 7, 15, and 11,



**Figure 3.4.** Syntax for declarations.

The amount of memory required for a variable depends on its type and the characteristics of the local computer system. Normally, simple variables are 1 to 8 bytes long. For example, an `int` (in many machines) uses four bytes of memory while a `double` occupies eight, and a `char` only one<sup>4</sup>.

**Declarations.** A variable is created by a **declaration**. (Unlike some languages, C requires every variable name to be declared explicitly.) The declaration specifies the type of object that is needed, supplies a name for it, and directs the compiler to allocate space for it in memory. Use of a variable is restricted to the statements within that block that declares it. We say such variables are **local to the block** because they are not “visible” to other blocks.

Declarations can appear anywhere in your code<sup>5</sup>. Most declarations are written at the beginning of a block of code, just after the opening `{`. This placement makes the declarations easy to locate. Sometimes, however, we declare a variable in the middle of a block, especially when it is going to be used to control a loop or a switch<sup>6</sup>.

**Syntax for declarations.** The simplest declaration is a type name followed by a variable name, ending with a semicolon. The following declaration creates an **integer variable** named `minutes`:

```
int minutes;
```

For each name declared, the compiler will create a storage object by allocating the right amount of memory for the specified type of data. The address of this storage object becomes associated with or *bound* to the variable name. Thereafter, when the programmer refers to the name, the compiler will use the associated address. Sometimes we need to refer explicitly to the address of a variable; in those contexts, we write an ampersand (which means “address of”) in front of the variable name. For example, `&minutes` means “the address of the variable `minutes`.”

The general form of a declaration, shown in Figure 3.4, contains four parts: zero or more modifiers,<sup>7</sup> a type name, an object name or names, and an optional initializer for each name. The parts are written in the order given, followed by a terminating semicolon, as diagrammed in Figure 3.4. The various different data types will be explored in the next few chapters. Object naming conventions are discussed shortly. An **initializer** gives a variable an initial value. This is done by using an `=` sign, followed by a value of the appropriate type.<sup>8</sup>

If multiple objects are declared, and possibly initialized, using a single statement, the objects are separated by commas. We have adopted a style in which most declaration lines declare one variable and have a comment explaining the use of that variable. This is a good way to write a program and make it self-documenting.

<sup>4</sup>A full treatment of types `int` and `double` is given in Chapter 7 where we deal with representation, overflow, and the imprecise nature of floating point numbers.

<sup>5</sup>The C99 standard loosened the restrictions on placement of declarations.

<sup>6</sup>Loops and switches will be covered later in this chapter and in Chapter 6.

<sup>7</sup>In the C standard, the properties `const` and `volatile` are called *type qualifiers* and `extern`, `static`, `register`, and `auto` are called *storage class specifiers*. For simplicity, and because the distinctions are not important here, we refer to both as *modifiers*. The modifiers are optional and, with the exception of `const`, you need not understand or use them for the time being. We use `volatile` in Chapter 15. We discuss `static` in Chapter 10 and all storage classes, in general, in Chapter 19. Additionally, the modifier `extern` is important in complex, multimodule programs and is demonstrated in Chapter 20.3.

<sup>8</sup>The rules for initializers will be given as each type is considered.

Four variables are declared here and diagrammed below.

```
int main( void )
{
    double length;           // Length, in meters (uninitialized).
    double weight = 1.5;     // Weight, in kilograms (initialized to 1.5).
    int k, m = 0;            // One uninitialized and one initialized variable.
    char gender = 'F'        // Use a char literal to initialize a char variable.
    ...
}
```




---

Figure 3.5. Simple declarations.

**Declaration examples.** Two examples of variable declarations were given in Chapter 2, Figure 2.7, which declares four `double` variables, `n1`, `n2`, `n3` and `average`. Figure 3.5 illustrates variations on the basic declaration syntax and shows how to draw diagrams of variables. The first line declares one variable of type `double`, as we have done many times already. This line tells C to allocate enough memory to store a `double` value and use that storage location whenever the name `length` is mentioned. The declaration does not put a value into the variable; it will contain whatever was left in that storage location by the program that previously ran on the computer. This value is unpredictable and meaningless in the present context. Formally, we say that it is an **undefined value**; informally, we call it **garbage**. The garbage stays there until a value is read into the variable (perhaps by `scanf()`) or stored in the variable by an assignment statement.

A variable is diagrammed, as in Figure 3.5, by drawing a box with the variable name just above the box and the current value inside it. The size of each box is proportional to the number of bytes of storage required on a representative C system. In the diagram of `length`, its undefined value is represented by a question mark. Variable diagrams, or **object diagrams**, are a concrete and visual representation of the abstractions that the program manipulates. We use them to visualize relationships among objects and to perform step-by-step traces of program execution. Object diagrams become increasingly important when we introduce compound data objects such as arrays, pointers, and data structures.

The second line in Figure 3.5 declares the variable `weight` and gives it an initial value. This shorthand notation combines the effect of a declaration and an assignment:

```
double weight;      // Weight, in kg. Contains garbage.
...
weight = 1.5;       // Now weight contains value 1.5
```

As we progress to more complex programs, the ability to declare and initialize a variable in one step will become increasingly important.

The third line in Figure 3.5 declares two variables, `k` and `m`, and it initializes `m`. It does not initialize `k`. To do that, another `=` sign and value would be needed prior to the comma. This omission is a common mistake of beginning programmers. Combining two declarations in this manner on one line saves space and writing effort. However, it does not provide a place to put separate comments explaining the purpose of the variables and so should be done only when the meaning of the variables is the same or they are logically related.

**Caution.** C has a somewhat arbitrary restriction that all of the declarations in a program block must come immediately after the `{` that follows `main( void )` and before any statements. (This restriction is relaxed in C++.)

**Assignment.** An assignment statement is one way to put a value into a variable. The term *variable* is used because a program can change the value stored in a variable by assigning a new value to it. An **assignment** begins with the name of the variable that is to receive the value. This is followed by an `=` sign and a value to be stored (or an arithmetic expression that computes a value, see later). As an example,

```
minutes = 10;      // Store value 10 in variable "minutes".
```

### 3.3.3 Constants and Literals

**Literals.** In addition to variables, most programs contain **constants**, which represent values that do not change. A **literal constant** is one that is written, literally, in your code; in the assignment statement `radius = diameter/2.0;` 2.0 is a literal **double**. For each type of value in C, there is a way to write a literal constant. These will be explained as we go through the various types. Here is a table of literal formats for the three types we have covered

C Syntax for Literal Constants

Type	Examples	Notes
<code>double</code>	3.1, .045	A string of digits with a decimal point.
<code>double</code>	4.5E-02	.045 written in scientific notation.
<code>int</code>	32767	A string of digits with an optional sign; no decimal point, no commas.
<code>char</code>	'A' or '%'	A single character enclosed in single quotes.

**Symbolic constants.** A **symbolic constant** is a constant to which we have given a name. There are two ways in C to create a symbolic constant: a `#define` command and a `const` declaration. Physical constants such as  $\pi$  often are given symbolic names. We can define the constant PI as

```
#define PI 3.1416
```

The `#define` commands usually are placed after the `#include` commands at the top of a program. It is sound programming practice to use `#define` to name constants rather than to write literal constants in a program, especially if they are used more than once in the code. This practice makes a program easier to debug because it is easier to locate and correct one constant definition than many occurrences of a literal number.

Note three important syntactic differences between `#define` commands and assignment statements: (1) the `#define` command does not end in a semicolon while the assignment statement does; (2) there is no `=` sign between the defined constant name and its value, whereas there always is an `=` sign between the variable and the value given to it; and (3) you cannot use the name of a constant on the left side of an assignment statement, only a variable.

Often, using symbolic names rather than literal constants saves trouble and prevents errors. This is true especially if the literal constant has many digits of precision, like a value for  $\pi$ , or if the constant might be changed in the future. Well-chosen descriptive names also make the program easier to read. For example, we might wish to use the temperature constant, -273.15, which is absolute zero in degrees Celsius. A name like `ABSO_C` is more meaningful than a string of digits.

**Using `#define`.** A `#define` command is generally used to define a physical constant such as `PI`, `GRAVITY` (see Figure 3.7) or `LITR_GAL` (see Figure 3.18), or to give a name to an arbitrarily defined value that is used throughout a program but might have to be changed at some future time. (The first example of such usage is the loop limit `N` in Figure 6.15.) We use a `#define` for an arbitrary constant so that changing the constant is easy if change becomes necessary. Changing one `#define` is easier than changing several references to the constant value, and finding the `#define` at the top of the file is easier than searching throughout the code for copies of a literal constant.

When you use a name that was defined with `#define`, you actually are putting a literal into your program. Commands that start with `#` in C are handled as a separate part of the language by a part of the compiler, called the *preprocessor*, that looks at (and possibly modifies) every line of source code before the main part of the compiler gets it. The preprocessor identifies the `#define` commands and uses them to build a table of defined terms, with their meanings. Thereafter, each time you refer to a defined symbol, the preprocessor removes it from your source code and replaces it by its meaning. The result is the same as if you wrote the meaning, not the symbol in your source code. For example, suppose a program contained these two lines of source code before preprocessing:

```
#define PI 3.1415927
area = PI * radius * radius;
```

After preprocessing, the `#define` is gone. In some compilers, you can instruct the compiler to stop after preprocessing and write out the resulting code. If you were to look at the preceding assignment statement in that code, it would look like this:

```
area = 3.1415927 * radius * radius;
```

Note the following things about a `#define` command:

- No `=` sign appears between the constant's name and its value.
- The line does *not* end in a semicolon, and you do *not* write the type of the name. (The C translator will deduce the type from the literal.)
- In this example, PI is a `double` constant because it has a decimal point.

The C preprocessor always has been a source of confusion and program errors. However, these difficulties are caused by advanced features of the preprocessor, not by simple constant definitions like those shown here. A beginning programmer can use `#define` to name constants with no difficulty. The programmer usually is unaware of the substitutions made by the preprocessor and does not see the version with the literal constants.

**The `const` qualifier.** You also can create a constant by writing the `const modifier` at the beginning of a variable declaration. This creates an object that is like a variable in every way except that you cannot change its value (so it is not really like a variable at all).<sup>9</sup>. A `const` declaration is like a variable declaration except that it starts with the keyword `const` and it must have an initializer.

A symbolic name is one way to clarify the purpose of a constant and the meaning of the statement that uses the constant. That C provides two different ways to give a symbolic name to a constant value (`#define` and `const`) might seem strange. C does so because each kind of constant can be used to do some advanced things that the other cannot. We recommend the following guidelines for defining constants:<sup>10</sup>

- Use `#define` to name constants of simple built-in types.
- Use `const` to define anything that depends on another constant.

This usage is illustrated in Figure 3.6. The lines in this figure are excerpted from a program that computes a payment table for a loan. The program uses `#define` for the annual interest rate, because while it is constant for this particular loan, it is likely to change for future loans. By placing the `#define` at the top of the program, we make it easy to locate and edit the interest rate at that future time. The monthly rate is one-twelfth of the annual rate; we declare it as a `const double` initialized to `RATE/12`. We use `const` rather than `#define` because the definition involves a computation.

The loan amount and the monthly interest are defined as variables, because they decrease each month. The monthly payment normally will be \$100 but we do not define it as a constant because the loan payment on the final month will be smaller.

Following the declarations in the figure are diagrams for the objects declared. The variables `payment`, `loan`, and `interest` are diagrammed as variable boxes. The striped box indicates that `mrate` is a constant variable and cannot be changed. No box is drawn for `RATE` because it is implemented as a literal. The lower line in the diagram shows the changes in the variables produced by the two assignment statements.

### 3.3.4 Names and Identifiers

As a programmer works on the problem specification and begins writing code to solve a problem, he or she must analyze what variables are needed and invent names for them. No two programmers will do this in exactly the same way. They probably will choose different names, since naming is wholly arbitrary. They might even use different types of variables or a different number of variables, since the same goals can often be accomplished in many ways. In this section, we introduce guidelines and formal syntactic rules for naming objects.

The technical term for a name is an **identifier**. You can use almost any name for an object, subject to the following constraints. These are absolute rules about names:

1. It must begin with a letter or underscore.
2. The rest of the name must be made of letters, digits, and underscores.
3. You cannot use C keywords such as `double` and `while` to name your own objects. You also should avoid the names of functions and constants in the C library, such as `sin()` and `scanf()`.
4. C is case sensitive, so `Volume` and `volume` are different names.

---

<sup>9</sup>A `const` variable has an address. This becomes important in advanced programs and in C++.

<sup>10</sup>These guidelines are consistent with the advanced uses of constants and with usage in C++.

```
#define RATE .125           // Annual interest rate.
const double mrate = RATE/12; // Monthly interest rate.
double payment = 100.00;    // Monthly payment.
double loan = 1000.00;     // Remaining unpaid principle amount.
double interest;          // Current month's interest.
```

	mrate			
constants:	0.01042	initial values	payment	loan
RATE: .125		of variables:	100.00	1000.00

```
interest = mrate * loan ;
loan = loan + interest - payment;
```

	payment	loan	interest
variables after assignments:	100.00	910.4166	10.4166

Figure 3.6. Using constants.

5. Some compilers limit names to 31 characters. Very old C compilers use only the first eight characters of a name.

These are guidelines for names:

1. Use one-letter names such as `x`, `t`, or `v` to conform to standard engineering and scientific notation. Writing `d = r * t` is better for our purposes than writing the lengthier `distance = rate * time`. Otherwise, avoid single-letter names.
2. When you have two similar quantities, such as two time instances, you might call them `t1` and `t2`. Otherwise, avoid using such similar names.
3. Use names of moderate length. Most names should be between 2 and 12 letters long.
4. Avoid names that look like numbers; `0`, `1`, and `I` are very bad names.
5. Use underscores to make compound names easier to read: `tot_vol` or `total_volume` is clearer than `totalvolume`.
6. Try to invent meaningful names; `x_coord` and `y_coord` are better names than `var1` and `var2`.
7. Do not use names that are very similar, such as `metric_distance` and `meter_distance` or `my_var` and `my_varr`.

### 3.3.5 Arithmetic and Formulas

Arithmetic formulas, which are called **expressions**, are written in C in a notation very much like standard mathematical notation, using the **operators** `+` (add), `-` (subtract), `*` (multiply), and `/` (divide). Normal mathematical operator **precedence** is supported; that is, multiplication and division will be performed before addition and subtraction. As in mathematics, parentheses may be used for grouping<sup>11</sup>. These operators are combined with variable names (such as `radius`), constants (such as `PI`), or **literal** values such as `3.14` or `10` to form expressions. The result of an expression can be used for output or it can be stored in memory by using assignment. For example, the following statement computes the area of a circle with radius `r` and stores the result in the variable `area`:

<sup>11</sup>Many more operators and the details of operator precedence and associativity are given in Chapter 4

```
area = 3.1416 * r * r;
```

Using a constant definition (`#define PI 3.1416`) permits us to rewrite the area formula thus:

```
area = PI * r * r;
```

## 3.4 Simple Input and Output

A call on an input function is one way to put a value in a variable; we call an output function when we want to see the value stored there. The input and output facilities in C are some of the most complex parts of the language, yet we need to use them as part of even the simplest programs. The best way to start is to learn to do input and output in a few simple ways. In this chapter, we focus on the elementary use of just three I/O functions:

Function name	Meaning of name	Purpose of function
<code>puts()</code>	Put string and newline	To write a message and a newline on the screen
<code>printf()</code>	Print output using a format	To write messages and data values.
<code>scanf()</code>	Scan input using a format	To read a data value from the keyboard

These functions (and many others) are in the standard input/output library (`stdio`), which is “added” to the program when you write `#include <stdio.h>`. Several examples given in this chapter use these functions with the hope that you can successfully imitate them.<sup>12</sup>

**Streams and buffers.** Input and output in C are handled using **streams**. An *input stream* is a device-independent connection between a program and a specified source of input data; an *output stream* connects a program to a destination for data. In this book, we use streams connected to data files or devices such as the computer’s keyboard and monitor screen. Three streams are predefined in standard C: one for input (`stdin`), one for output (`stdout`), and one for error comments (`stderr`). The standard output stream is directed by default to the operator’s video screen but can be redirected to a printer or a file.<sup>13</sup> The standard input stream normally is connected to the keyboard but also can be redirected to get information from a file.

Input and output devices are designed to handle chunks of data. For example, a keyboard delivers an entire line of characters to the computer when you hit Enter, and a hard disk is organized into clusters that store 1,000 or more characters. When we read from or write to a disk, the entire cluster is read or written. On the other hand, programs typically read data only a few numbers at a time and write data one number or one line at a time. To bridge the gap between the needs of the program and the characteristics of the hardware, C uses buffers. Every stream has its own **buffer**, an area of main memory large enough to hold a quantity of input or output data appropriate for the stream’s device.

When the program uses `scanf()` to read data, the input comes out of the buffer for the `stdin` stream. If that buffer is empty, the system will stop and wait for the user to enter more data. If a user types more data than are called for, the extra input remains in the input buffer until the next use of `scanf()`.

Similarly, when the program uses `printf()` to produce output, that output goes to the output buffer and stays there until the program prints a newline character (denoted by `\n`) or until the program stops sending output and switches to reading input. This permits a programmer to build an output line one number at a time, until it is complete, then display the entire line. However, the most common use of `printf()` is to print an entire line at one time.

### 3.4.1 Formats.

Some input and output functions, like `puts()`, read or write a single value of a fixed type. A call on `puts()`, which outputs a single string, is very simple: We write the message enclosed in double quotes inside the parentheses following the function name. For example, we might write this lines as the first statement in a program that computes square roots:

<sup>12</sup>A note about notation: In writing about C functions, we often use the name of a function separately, outside the context of program code. At these times, it is customary to write an empty pair of parentheses after the function name. The parentheses remind us that we are talking about a function rather than some other kind of entity. In a program, they distinguish a function call (with parentheses) and a reference to a function (without).

<sup>13</sup>The complexities of streams will be explored in Chapter 14.

```
puts( "Compute the square root of a number." );
```

Other input and output functions, including `scanf()` and `printf()`, can read or write a list of values of varying types. Using these functions is more complex than using `puts()` because two kinds of things may be written inside the parentheses. First comes a **format** string, which describes the form of the data. It describes how many items will be read or written and the type of each item. Following that is either a list of addresses for the input data (for `scanf()`) or a list of expressions whose results are to be printed (for `printf()`). The complete set of rules for formats is long and detailed.<sup>14</sup> Fortunately it is not necessary to understand formats fully in order to use them. In this chapter, we show how to write simple formats for three types of data.<sup>15</sup>

A format is a string (a series of characters enclosed in double quotes) that describes the form and quantity of the data to be processed. Input and output formats are quite different and will be described separately. Both, however, contain conversion specifiers. The **conversion specifiers** in the format tell the input or output function how many data items to process and what type of data is stored in each item. Each conversion specifier starts with a percent sign and ends with a code letter(s) that represents the type of the data.

Data Type	Input Specifier	Output Specifier	Notes
char	"%c"	"%c"	Type a space between the quote and the % sign for input.
int	"%i"	"%i"	"%d" also works.
double	"%lg"	"%g"	Use %lg for input, but only %g" (without the letter 1) for output.

**Using input formats.** An input format is a series of conversion specifiers enclosed in quotes; for example, the format string "%i" could be used to read one integer value and "%i%i" could be used to read two integers that are separated by spaces on the input line. In a call on `scanf()`, the format is written first, followed by a list of the addresses of the variables that will receive the data after they are read. Here are two complete calls on `scanf()`:

```
scanf( "%i", &minutes );
scanf( " %c%i%lg", &gender, &age, &weight );
```

The first line tells `scanf()` to read one integer value and store it at the memory location allotted to the variable named `minutes`. The second line tells `scanf()` to read a character an integer, and a real number. The character will be stored at the address of the variable named `gender`, the integer in the variable named `age`, and the real number in the variable named `weight`.

**Using output formats.** Output formats contain both conversion specifiers and words that the programmer wishes to see interspersed within the data. Therefore, output formats are longer and more complex than input formats. An appropriate output format to print the data just read might look like this:

```
"Gender: %c Age: %i Weight: %g\n"
```

The words and the spaces are written exactly the way they should appear on the screen. The `%c`, `%i` and `%g` tell `printf()` where to insert the data values in the sentence. The `\n` represents a newline character. We need it with `printf()` to cause the output cursor to go to a new line. (The `\n` is not needed with `puts()`.) Hence, most `printf()` formats end in a newline character. Make this a habit: *Use \n at the end of every format string to send the information to your screen immediately and prepare for the next line.*<sup>16</sup>

Following the format string in a call on `printf()` is the list of variables or expressions whose values we want to write (do not use the ampersand for output). Exactly one item should appear in this list for each `%` in the format. A complete call on `printf()` might look like this:

```
printf( "Gender: %c Age: %i Weight: %g\n", gender, age, weight );
```

<sup>14</sup>These rules can be found in any standard reference manual, such as S. Harbison and G. Steele, *C: A Reference Manual*, 4th ed (Englewood Cliffs, NJ: Prentice-Hall, 1995).

<sup>15</sup>As other types of data are introduced in Chapters 7 through 11, more details about formats will be presented.

<sup>16</sup>The exception is a format string used to display a user prompt. These normally end in a colon and a space so that the screen cursor does not move to a new line and the user types the input on the same line as the prompt.

This tells `printf()` to print the values stored in the variables `gender`, `age`, and `weight`. These values will appear in the output after labels that tell the meaning of each number. The output from this line might be

```
Gender: M  Age: 21  Weight: 178.5
```

## 3.5 A Program with Calculations

Now you know how to define constants and variables, how to get letters and numbers into and out of the computer’s memory, and how to do simple calculations. You know, in general, the form of a program. In this section, we combine all these elements in a simple program in Figure 3.7 to illustrate variable declarations, assignments, input, output, and calculations. In this program, some parts of the code are boxed. Program notes corresponding to these boxes are given below.

**The problem.** A grapefruit is dropped from the top of a very tall building<sup>17</sup>. It has no initial velocity since it is dropped, not thrown. The force of gravity accelerates the fruit. Determine the velocity of the fruit and the distance it has fallen after `t` seconds. The time, `t`, is read as an input.

**Notes on Figure 3.7: Grapefruits and gravity.**

*Introductory comments.*

- The first two lines of “Grapefruits and Gravity” are a comment block. The first line of such a block starts with `/*` and the last line ends with `*/`. Other lines in the block do not need any special mark at the beginning, but using `**` or `//` (as shown here) makes an attractive heading.
- Good programmers put comments at the top of a program to identify the program, its author, and the date or version number. Comments are also written throughout the code to explain the purpose of each group of statements. The compiler does not use the comments—we write them for the benefit of the humans who will read the code.

*First box: preprocessor commands.*

- Any line that starts with a `#`, called a *preprocessor command*, is handled by the preprocessor before the compiler translates the code.
- In this program, as in most, we ask the preprocessor to bring in the file `stdio.h` and include the contents of that file as part of the program. This is the header (basically, the table of contents) for the standard I/O library. This command makes the standard input-output library functions available for use.
- We use a `#define` command to give a symbolic name to the constant for the acceleration of gravity at sea level.

*Second box: the declarations.*

- Declarations are used to define the names of variables that will be used in statements later in the program.
- We declare three `double` variables named `t`, `y`, and `v`. This instructs the compiler to allocate enough space in memory to store three real values and use the appropriate location every time we refer to `t`, `y`, or `v`. On most common machines, 8 bytes of memory will be allocated for each variable.
- Note that `t`, `y`, and `v` all are variables of type `double`. When you have several variables of the same type, you also declare them on separate lines, as shown, or you may declare them all on one line like this: `double t, y, v;`. In this problem, we declare one variable per line so that there is space for a comment that explains the meaning or purpose of each variable. This is good programming practice.

*Third box: printing an output title.*

- A well-designed program displays a title and a greeting message so that the user knows that execution has started.
- We could use either `puts()` or `printf()` to print the title, but if we use `printf()` we must end the string with a newline character if we want the output cursor to move down to the next line.
- Here, we print a 3-line title, using three lines of code. Where we break the code, we end the line with a quote mark and start the next with indentation and a quote mark.
- Program execution begins with the first box following the declarations, proceeding sequentially through the other boxes to the end of the code. This is studied more fully in the next section.

---

<sup>17</sup>This problem will serve as the basis for a series of example programs in the following sections.

***Fourth box: user input.***

- A **prompt** is a message displayed on the video screen that tells the user what to do. A program must display a prompt whenever it needs input from the human user.
- In this prompt, we ask the user to type in the time it took the grapefruit to hit the ground. We *do not* put a newline character at the end of the message because we want the input to appear on the same line as the prompt. We *do* leave a space after the colon to make the output easy to read. The user sees

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.
```

```
Input seconds: 10
```

- In the call on `scanf()`, we send two pieces of information to the `scanf()` function: the format string and the address of the variable to receive the input. This causes the system to scan the `stdin` input stream to find and read a value for `t` (time, in seconds). This value may be entered with or without a decimal point.
- The `%` sign in the format is the beginning of a conversion specifier. It tells us to read one item and specifies the type of that item. We use `%lg`, for “long general-format real,” to read a value of type `double`. (The

```
// Determine the velocity and distance traveled by a grapefruit
// with no initial velocity after being dropped from a building.
//

#include <stdio.h>
#define GRAVITY 9.81           // gravitational acceleration (m/s^2)

int main( void )
{
    double t;    // elapsed time during fall (s)
    double y;    // distance of fall (m)
    double v;    // final velocity (m/s)

    printf( "\nWelcome.\n"
            " Calculate the height from which a grapefruit fell\n"
            " given the number of seconds that it was falling.\n\n" );

    printf( " Input seconds: " ); // prompt for the time, in seconds.
    scanf ( "%lg", &t );        // keyboard input for time

    y = .5 * GRAVITY * t * t;      // calculate distance of the fall
    v = GRAVITY * t;                // velocity of grapefruit at impact

    printf( "     Time of fall = %g seconds \n", t );
    printf( "     Distance of fall = %g meters \n", y );
    printf( "     Velocity of the object = %g m/s \n", v );

    return 0;
}
```

Figure 3.7. Grapefruits and gravity.

letter between the % and the g is a lowercase *letter l*, not a numeral 1.)

- After the format comes the address for storing the input data. In this case, the data will be stored at &t, the address of the variable t.

#### *Fifth box: calculations.*

- The symbol = is used to store a value into a variable. Here we store the result of the calculation .5 \* GRAVITY \* t \* t, which is the standard equation for distance traveled under the influence of a constant force after a given time, in the variable named y.
- The calculation for height, y, uses several \* operators. The \* means “multiply.” When there is a series of \* operators, they are executed left to right.
- The second formula is standard for computing the terminal velocity of an object with no initial velocity under the influence of a constant force.

#### *Sixth box: the output.*

- First, the input value (time) is echoed onto the video screen to convince the user that the calculations were done with the correct value.
- Two `printf()` statements write the answers to the screen. We use a %g specifier to write a `double` value. Note that this is different from a `scanf()` format, where we need %lg for a `double`.
- The `printf()` function leaves the output cursor on the same output line unless you put a newline character, \n, in the format string. In this case, we use a \n to put the velocity value on a different line from the distance.
- This would be a typical output:

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 10
    Time of fall = 10 seconds
    Distance of fall = 490.5 meters
    Velocity of the object = 98.1 m/s

Gravity has exited with status 0.
```

#### *Last box: termination.*

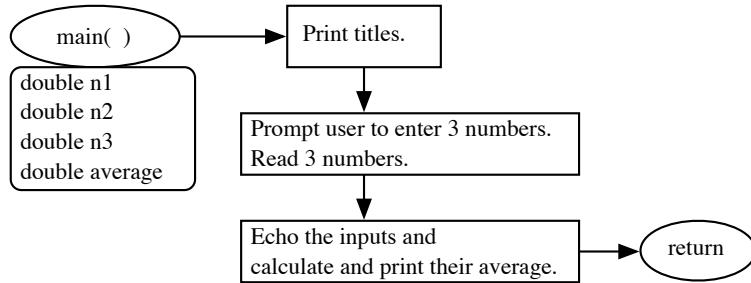
- It is sound programming practice to display a termination comment. This leaves no doubt in the mind of the user that all program actions have been completed normally.
- At the top of every program, we write `int main( void )`. This tells the operating system to expect to receive a termination code from your program.
- The last line in every program should be a `return 0;` statement. The zero is a termination code that tells the system that termination was normal.

## 3.6 The Flow of Control

All the examples we have considered so far execute the code (instructions) line by line from beginning to end. However, in most practical programs, it is necessary to follow alternative paths of execution, depending on the data values and other conditions. For example, suppose you want to calculate the velocity of an object, as in Figure 3.7, but you want to prohibit cases that make no physical sense; that is, negative values of time. Two potential courses of action could be taken: (1) Do the calculation and display the answer or (2) comment on the illegal input data and skip the calculation. Another reason we need nonsequential execution is to enable a program to analyze many data values by repeating one block of code.

When a C program is executed, action starts at the first statement following the declarations. For example, in Figure 3.7, execution starts with the `printf()` statement. From there, execution proceeds to the next statement and the next, in order, until it reaches the `return` statement at the program, at which point control returns to the operating system. This is called **simple sequential execution**. *Control statements* are used to create conditional branching and repetitive paths of execution in a program. *Flow diagrams* are used as graphical illustrations of the execution paths that these control statements create.

This is a flow diagram of the program in Figure 2.7. It illustrates simple straight-line control and the way we begin and end the diagram of a function.



**Figure 3.8. A complete flow diagram of a simple program.**

A **flow diagram** is an unambiguous, complete diagram of all possible sequences in which the program statements might be executed. We use flow diagrams to illustrate the flow of control through the statements of a program or part of a program. This is not so necessary for simple sequential execution. However, a two-dimensional graphic representation can greatly aid comprehension when control statements are used to implement more complex sequencing. A few basic rules for flow diagrams follow and are illustrated in Figure 3.8:

1. A flow diagram for a complete program begins with an oval “start” box at the top and ends with an oval return box at the bottom.
2. Declarations need not be diagrammed unless they contain an initializer (used to give the variable a starting value and discussed more in the next chapter). It is probably clearer, though, if you include one round-cornered box below the start oval that lists the variables declared within the function.
3. The purpose of each statement is written in the appropriate kind of box, each depending on the nature of the action. Since the purpose of the diagram is to clarify the logic of the function, we use English or pseudocode, not C, to describe the actions of the program. Such a diagram could be translated into languages other than C.
4. Simple assignment statements and function calls are written in rectangular boxes. Several of these may be written in the same box if they are sequential and relate to each other.
5. Arrows connect boxes in the sequence in which they will be executed, from start to finish. In the diagram of a complete function, no arrow is left dangling in space and no box is left unattached. All arrow heads must end at a box of some sort, except in the diagram of a program fragment, where the beginning and ending arrows might be left unattached.
6. The diagrams are laid out so that flow generally moves down or to the right. However, you may change this convention if it simplifies your layout or makes it clearer.
7. No arrow ever branches spontaneously. Every tail has exactly one head.

As another example, Figure 3.9 shows the diagram of the program in Figure 3.7. The graph consists of five nodes connected by arrows that indicate the flow of control during execution:

1. A start oval at the top, attached to a box listing the variable declarations.
2. A rectangular box listing statements that print a title and prompt for and read the input.
3. A box that calculates the values for distance and velocity using appropriate formulas.
4. A box that echoes the input and outputs the answers.
5. A return oval at the bottom that terminates the program.

Note that these boxes correspond roughly to the boxed units of code in the program.

This is a flow diagram of the program in Figure 3.7.

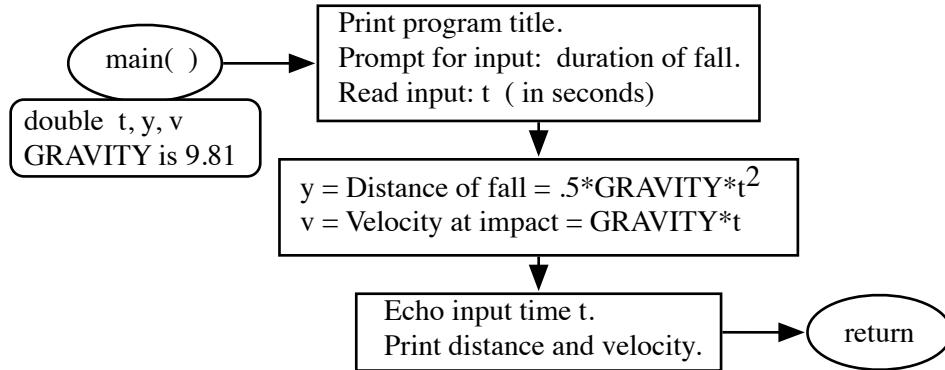


Figure 3.9. Diagram of the grapefruits and gravity program.

## 3.7 Asking Questions: Conditional Statements

A large part of the power of a computer is the ability to take different actions in different situations. For this purpose, all computers have instructions that do various kinds of conditional branching. These instructions are represented in C by the `if` and `if...else` statements.

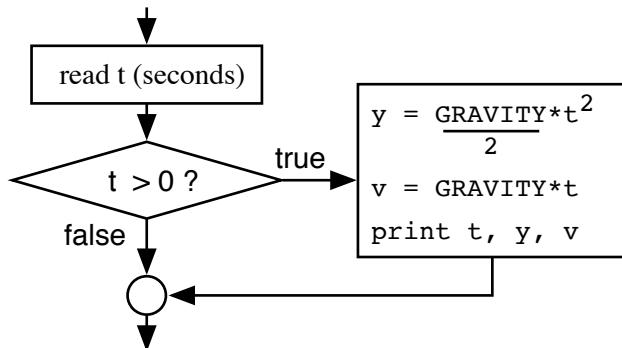
### 3.7.1 The Simple if Statement

In a simple `if` statement, the keyword `if` is followed by an expression in parentheses, called the *condition*, followed by a single statement or a block of statements in braces, called the *true clause*. At run time, the expression is evaluated and its result is interpreted as either true or false. If true, the block of statements following the condition is executed. If false, that block of statements is skipped. Execution continues with the rest of the program following the conditional.

This control pattern is illustrated by the program in Figure 3.10. We use a simple `if` statement here to test whether the input data make sense (the input is a valid time value). If so, we process the data; if not, we do nothing.

#### Notes on Figure 3.10. Asking a question.

1. The outer box contains the entire simple `if` statement, which consists of
  - The keyword `if`.
  - The condition ( $t > 0$ ).
  - A block (inner box) containing two assignment statements and three `printf()` statements.
2. As shown in the flow diagram, below, there is only one control path into the `if` unit and one path out. Control branches at the `if` condition but rejoins immediately below the true clause.



3. We use an `if` statement here to test whether the input data makes sense (the time is positive). If so, we follow the `true` path in the diagram and process the data. if not, we follow the `false` path in the diagram and do nothing.

In either case, the next statement executed will be the final `puts()`. A sample output from the `false` path follows. Note that the user gets no answers at all and no explanation of why. This is not a good human-machine interface; it will be improved in the next program example.

Welcome.

This solves the same problem as Figure 3.7, except that we screen out invalid inputs.

```

// -----
// Determine the velocity and distance traveled by a grapefruit
// with no initial velocity after being dropped from a building.
//
#include <stdio.h>
#define GRAVITY 9.81      // gravitational acceleration (m/s^2)

int main( void )
{
    double t;           // elapsed time during fall (s)
    double y;           // distance of fall (m)
    double v;           // final velocity (m/s)

    printf( " \n\n Welcome.\n"
            " Calculate the height from which a grapefruit fell\n"
            " given the number of seconds that it was falling.\n\n" );
    printf( " Input seconds: " ); // prompt for the time, in seconds.
    scanf( "%lg", &t );        // keyboard input for time

    if (t > 0) {           // check for valid data.

        y = .5 * GRAVITY * t * t; // calculate distance of the fall
        v = GRAVITY * t;         // velocity of grapefruit at impact
        printf( "     Time of fall = %g seconds\n", t );
        printf( "     Distance of fall = %g meters\n", y );
        printf( "     Velocity of the object = %g m/s\n", v );
    }
    return 0;
}
  
```

Figure 3.10. Asking a question.

```
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.
```

```
Input seconds: -1
```

```
Gravity has exited with status 0.
```

4. The inner box (the true clause) shows what happens when  $t$ , the time, is positive: we execute the statements that calculate and print the answers. A sample output from this path might be

```
Welcome.
```

```
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.
```

```
Input seconds: 10
```

```
Time of fall = 10 seconds
```

```
Distance of fall = 490.5 meters
```

```
Velocity of the object = 98.1 m/s
```

```
Gravity has exited with status 0.
```

### 3.7.2 The if...else Statement

An `if...else` statement is like a simple `if` statement but more powerful because it lets us specify an alternative block of statements to execute when the condition is false. It consists of

The keyword `if`.

An expression in parentheses, called the *condition*.

A statement or block of statements, called the *true clause*.

The keyword `else`.

A statement or block of statements, called the *false clause*.

At run time, the condition will be evaluated and either the true or the false clause will be executed (the other will be skipped), depending on the result.

The syntactic difference between an `if...else` and a simple `if` statement is that the true clause of an `if...else` statement is followed immediately by the keyword `else`, while the true clause of a simple `if` statement is not. There must not be a semicolon before the `else`.

One common use of the `if...else` statement is to validate input data, it often is necessary to perform more than one test. For example, one might need to test two inputs or test whether an input lies between minimum and maximum acceptable values. To do this, we can use a series of `if...else` statements, as demonstrated in Figure 3.11, which is an extension of the program in Figure 3.10. In this program, we make two tests using two `if...else` statements in a row. We ensure that the input both is positive and does not exceed a reasonable limit, in this case 60 seconds. Using `if...else` in this manner, we can test as many criteria as needed.

The flow diagram corresponding to this new version of the program is given in Figure 3.12. In this diagram, you can see the column of diamond shapes typical of a chain of `if` statements. The `true` clause actions form another sequence to the right of the tests in the diamonds, and the final `false` clause terminates the diamond sequence. All the paths come together in the bubble before the termination message.

#### Notes on Figure 3.11: Testing the limits.

##### *First box: maximum time.*

- This program checks for inputs that are too large as well as those that are negative.
- Since the upper limit is an arbitrary number, it might be necessary to change it in the future. To make such changes easy, we define this limit, `MAX`, at the top of the program and use the symbolic name in the code.

***Large outer box: the if...else chain.***

- Before computations are made, we inspect the data for two errors (first two inner boxes). We skip the remaining tests and the calculations if either error is discovered. This method of error handling avoids doing a computation with meaningless data, such as the negative value in the previous version of the program.
- If no errors are found, we execute the code in the third inner box.

***First inner box: negative input values.***

- As in the prior example, the input must be positive to correspond to physical reality. The first **if** statement handles this.
- The **true** clause of this **if** statement prints an error comment. Following that, control goes to the **puts()** statement at the bottom of the program, skipping over the **else if** test and the **else** clause.

This solves the same problem as Figure 3.10, except that we limit valid inputs to be less than 1 minute.

```
#include <stdio.h>
#define GRAVITY 9.81           // Gravitational acceleration (m/s^2)

#define MAX      60             // Upper bound on time of fall.

int main( void )
{
    sdouble t;      // elapsed time during fall (s)
    double y;       // distance of fall (m)
    double v;       // final velocity (m/s)

    printf( " \n\n Welcome.\n"
            " Calculate the height from which a grapefruit fell\n"
            " given the number of seconds that it was falling.\n\n" );
    printf( " Input seconds: " );   // prompt for the time, in seconds.
    scanf( "%lg", &t );          // keyboard input for time

    if (t < 0) {                // check for negative input
        printf( " Error: time must be positive.\n\n" );
    }

    else if (t > MAX) {         // Is time value too big?
        printf( " Error: time must be <= %i seconds.\n\n", MAX );
    }

    else { // Input is valid; calculate distance and velocity
        y = .5 * GRAVITY * t * t; // calculate distance of the fall
        v = GRAVITY * t;          // velocity of grapefruit at impact
        printf( "     Time of fall = %g seconds \n", t );
        printf( "     Distance of fall = %g meters \n", y );
        printf( "     Velocity of the object = %g m/s \n", v );
    }

    return 0;
}
```

Figure 3.11. Testing the limits.

This is a diagram of the program from Figure 3.11.

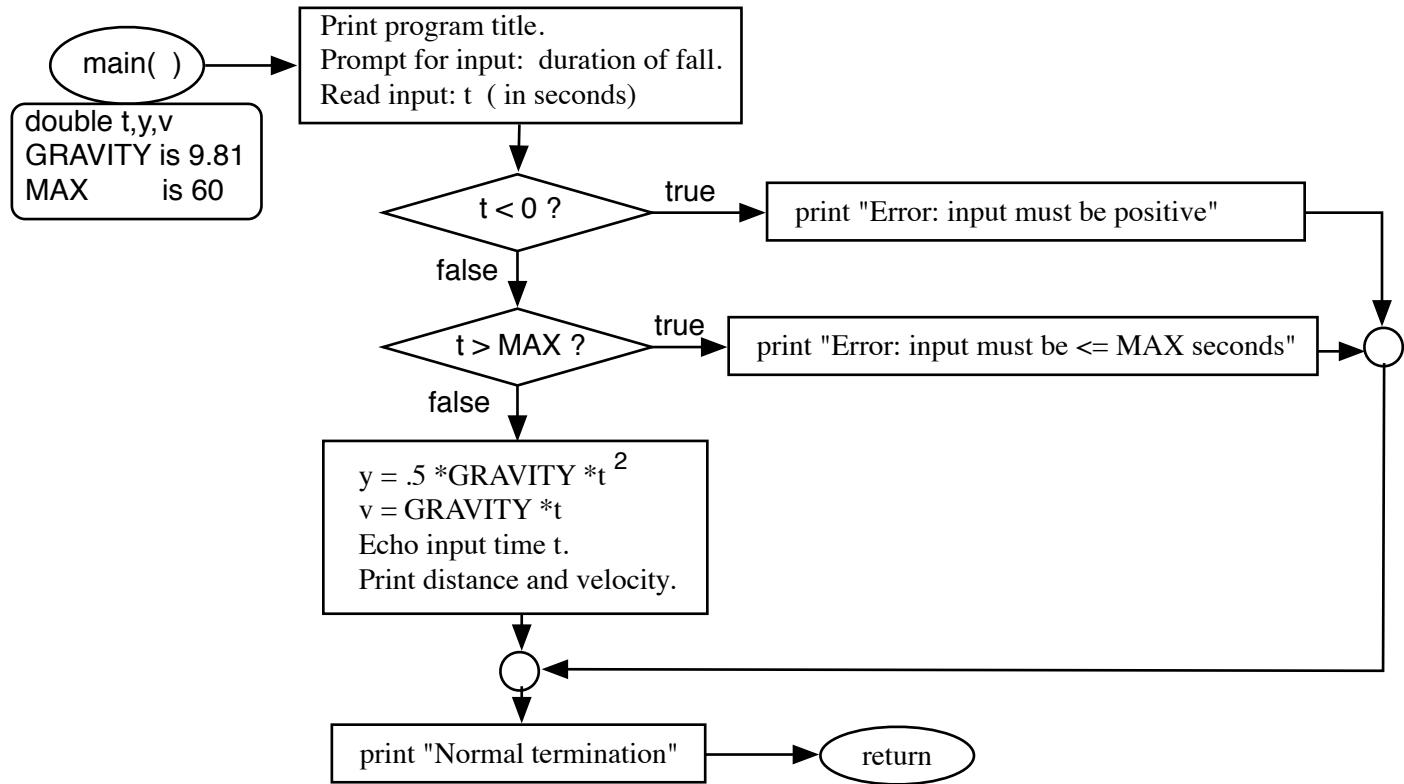


Figure 3.12. Flow diagram for Testing the limits.

#### Second inner box: *input values that are too large.*

- The elapsed time also must be reasonable. We compare the input value to a maximum allowable value, as defined in the problem specification (60 seconds in this case). The second `if` statement handles this.
- The `true` clause of this `if` statement prints an error comment. Then control skips the `else` clause and goes to the `puts()` at the bottom of the program.
- Here is a sample of the program's error handling:

```

Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 100
Error: time must be <= 60 seconds.
  
```

#### Third inner box: *the computation and output.*

- If the input seems valid, we calculate and print the distance and velocity using the given formulas. Then we echo the input and print the answers.
- Here is the output (without the titles) from a run with valid data:

```

Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.
  
```

```
Input seconds: 20
Time of fall = 20 seconds
Distance of fall = 1962 meters
Velocity of the object = 196.2 m/s
```

### 3.7.3 Which if Should Be Used?

We looked at three different ways to use `if` statements. These three control patterns are appropriate for different kinds of applications.

**The simple if statement.** The primary uses of the simple `if` statement:

- Processing a subset of the data items. Sometimes a collection of data contains some items that are relevant to the current process and others that are not of interest. We need to process the relevant items and ignore the others. For example, suppose that a file contains data from a whole census but the programmer is interested only in people over the age of 18. The entire file would need to be read but only a subset of the data items would be selected and processed.
- Handling data items that require extra work. Sometimes a few items in a data set require extra processing. For instance, a cash register program must compute the sales tax on taxable items but not on every item. It must test whether each product is taxable and, if so, compute and add the tax. If not, nothing happens.
- Testing for an error condition or goal condition within a loop and leaving the loop if that condition is found. This use of `if` with `break` will be discussed in Chapter 6.

**The if...else statement.** The primary applications of the `if...else` statement:

- Choosing one from a series of alternatives. Sometimes two or more alternative actions are possible and one must be selected. In this case, we often use a series of `if...else` statements to test a series of conditions and select the action corresponding to the first test whose result is `true`.
- Normal processing versus nonfatal error handling. In this control pattern, an input value is read and tested for legality. If it would cause a run-time error, an error message is given and the input is not processed in the usual way. This control pattern is illustrated by the outer box and diagram in Figure 3.11.
- Validating a series of inputs. We can use a series of `if...else` statements to test a series of inputs. The `true` clause of each statement would print an error comment, and the `false` clause of the last statement would process the validated data. This decision pattern is incorporated into Figure 3.11.

### 3.7.4 Options for Syntax and Layout (Optional Topic)

Normally each part of an `if` statement is written on a separate line and all the lines are indented except the `if`, the `else`, and the closing curly bracket. However, a C compiler does not care how you lay out your code.

**Indentation.** Inconsistent or missing indentation does not cause any trouble, it simply makes the program hard for a human being to read. Since the compiler determines the structure of the statement solely from the punctuation (semicolons and braces), an extra or omitted semicolon can completely change the meaning of the statement. Therefore, consistent style is important, both to make programs easier to modify and to help avoid punctuation errors.

**The condition.** The condition in parentheses after the keyword `if` can be any expression; it does not need to be a comparison. Whatever the expression, it will be evaluated. A zero result will be treated as false and a nonzero result will be treated as true. (Note, therefore, that any number except 0 is considered to be `true`.) The name of a variable, or even a call on an input function, is a legal (and commonly used) kind of condition.

**Very local variables.** Technically, we could declare variables inside any block, even one that is part of an `if` statement. However, this is not an appropriate style for a simple program.

---

With braces the code is spread out:

```
if (age > 18) {
    adults = adults + 1;           // Count the adults.
}
else {
    kids = kids + 1;             // Count the children.
}
```

Without braces the code is more compact:

```
if (age > 18)    adults = adults + 1; // Count the adults.
else            kids = kids + 1;   // Count the children.
```

---

**Figure 3.13.** The `if` statement with and without braces.

**Curly braces.** A `true` or `false` clause can consist of either a single statement or a block of statements enclosed in braces. If it consists of a single statement, the braces `{` and `}` may be omitted. This can make the code shorter and clearer if the resulting clause fits entirely on the same line as the keyword `if` or `else`. Figure 3.13 illustrates this issue. In it, the same `if` statement is written with and without braces. The first version is preferred by many experts, even though the braces are not required. However, others feel that the second version, without braces, is easier to read because it is written on one line as a single, complete thought. In either case, consistency makes code easier to read; a program becomes visually confusing if one part of an `if` statement has `{` and `}` and the other does not.

## 3.8 Loops and Repetition

The `if` statement lets us make choices. We test a condition and execute one block of code or another based on the outcome. Another kind of control structure is the loop, which lets us execute a block of statements any number of times (zero or more). Conditionals and loops are two of the three fundamental control structures in a programming language.<sup>18</sup> Loops are important because they allow us to write code once and have a program execute it many times, each time with different data. The more times a computation must be done, the more we gain from writing a loop to do it, rather than doing it by hand or writing the same formula over and over in a program.

C provides three types of loop statements that repeatedly execute a block of code: `while`, `do`, and `for`. The `while` statement is the most basic and is introduced first, in Figure 3.14.<sup>19</sup>

### 3.8.1 A Counting Loop

Every loop has a set of actions to repeat, called the **loop body**, and a loop test to determine whether to repeat those actions again or end the repetition. In this chapter, we study loops based on counting. In these loops, a variable is set to an initial value, then increased or decreased each time around the loop until it reaches a goal value. We call this variable a **loop counter** because it counts the repetitions and ends the loop when we have repeated it enough times. Our next example, shown in Figure 3.14, introduces a counting loop implemented using a `while` statement. It is a very simple program whose purpose is to make the repetition visible. The corresponding flow diagram used for `while` loops follows the code in the figure.

#### Notes on Figure 3.14: Countdown.

##### *First box: the loop variable.*

A counter is an integer variable used to count some quantity such as the number of repetitions of a loop.

##### *Second box: the loop.*

- Before entering a `while` loop, we must initialize the variable that will be used to control it. Here we scan an initial value into `days_left`, the counter variable.

---

<sup>18</sup>The third basic control structure, functions, will be introduced in Chapter 5.

<sup>19</sup>The other two loop statements will be presented in Chapter 6.

- A `while` statement has three parts, in the following order: the keyword `while`, a condition in parentheses, and a body. The loop body consists of either a single statement or a block of statements enclosed in braces, as shown in the program.
- To execute a `while` loop, first evaluate the condition. If the result is true, the loop body will be executed once and the condition will be retested.
- This sequence of execution is illustrated by the flow diagram at the bottom of Figure 3.14. In the diagram, the `while` loop is an actual closed loop. Control will go around this loop until the condition becomes false, at which point control will pass out of the loop. In this case, control will go to the `puts()` statement. The same diamond shape used to represent a test in an `if` statement is used to represent the loop termination test. The presence of a closed cyclic path in the diagram shows that this is a loop, rather than a conditional control statement.
- The statement `days_left = days_left - 1` tells us to use the old value of `days_left` to compute a new one. Read this expression as “`days_left` gets `days_left minus 1`” or “`days_left` becomes `days_left minus 1`”. (Do not call this assignment operation *equals*, or you are likely to become confused with a test for equality.) In detail, this statement means

---

This program demonstrates the concept of a counting loop. After inputting N, a number of days, it counts downward from N to 0.

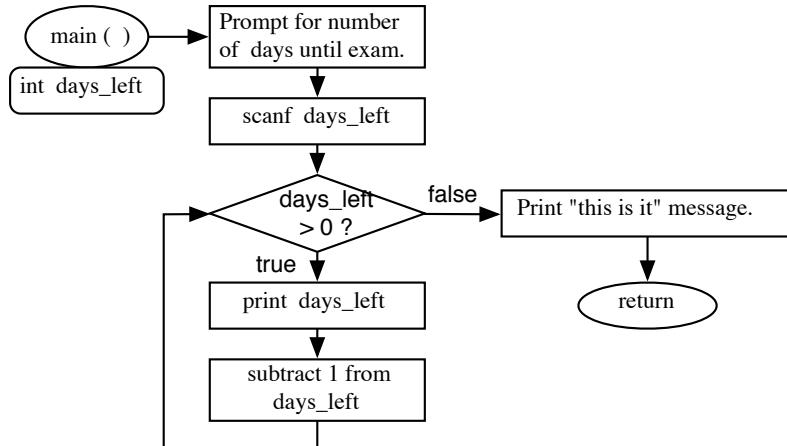
```
#include <stdio.h>

int main( void )
{
    int days_left; // The loop counter.

    printf( " How many days are there until the exam? " );
    scanf( "%i", &days_left); // Initialize the loop counter.

    while (days_left > 0) { // Count downward from 20 to 1.
        printf( " Days left: %i. Study now.\n", days_left );
        days_left = days_left - 1; // Decrement the counter.
    }

    puts( " This is it! I hope you are ready. " );
    return 0;
}
```




---

**Figure 3.14. Countdown.**

- Fetch the current value of `days_left`.
- Subtract 1 from it.
- Store the result back into the variable `days_left`.
- Each time we execute the body of the loop, the value stored in `days_left` will decrease by 1.
- Sample output:
 

```
How many days are there until the exam? 4
Days left: 4. Study now.
Days left: 3. Study now.
Days left: 2. Study now.
Days left: 1. Study now.
This is it! I hope you are ready.
```
- Control will leave the loop when the condition becomes false; that is, the first time that `days_left` is tested after its value reaches 0. This will happen before displaying the 0.

**Third box: after the loop.**

After leaving the loop, control goes to the statements that follow it. The call on `puts()` displays a message at the bottom of the output. The `return` statement returns to the operating system with a code of 0, indicating successful termination.

### 3.8.2 An Input Data Validation Loop

Interactive computer users are prone to errors; it is very difficult to hit the right keys all the time. It is common to lean on the keyboard or to hold a key down too long and type 991 instead of 91. A crash-proof program must be prepared for bad input, sometimes called *garbage input*. A much-repeated saying is “garbage in, garbage out” (GIGO); that is, the results of a program cannot be meaningful if the input was erroneous or illegal. Although it is impossible to identify all bad input, some kinds can be identified by comparing the data to the range of expected, legal data values. When an illegal data item is identified, the program can quit or ask for re-entry of the data.

A good interactive program uses input prompts to let the user know what specific inputs or what kinds of inputs are acceptable and what to do to end the process. If an input clearly is faulty, the program should (at a minimum) refuse to process it. Preferably, it should explain the error and give the user as many chances as needed to enter good data. One way to do this is the data validation loop.

A **data validation loop** (shown in Figure 3.15) prompts the user to enter a particular data item. It then reads the item and checks whether the input meets criteria for a reasonable or legal value. If so, the loop exits; if not, the user is informed of the error and reprompted. This continues until the user enters an acceptable value.

**Notes on Figure 3.15: Input validation using while.**

**First box: A valid odometer reading.**

- Before the beginning of a `while` data validation loop, the program must prompt the user for input and read it.
- The `while` loop tests the input. If it is good, the loop body will be skipped; otherwise, control enters the loop. Numbers entered must be small enough to be stored in an integer variable.
- The loop must print an error comment that indicates what is wrong with the input, reprompt the user, and read another input.
- Note that we need two prompts and two `scanf()` statements for this control pattern: one before the loop and another inside the loop.

**Second and third boxes: validating the other input.**

- Data validation loops all follow a pattern similar to the first loop. The major difference between the first two loops is that the mileage read by the first loop is used in the validation test of the second.
- In the third loop, a computation must be made before the data can be tested. Like the `scanf()` statement, this computation must be written twice, once before entering the loop and again at the end of the loop.

**The fourth box: correct data.** The box produced this output when correct data were supplied:

Miles Per Hour Computation

```
Odometer reading at beginning of trip: 061234
Odometer reading at end of trip: 061475
Duration of trip in hours and minutes: 4 51
Average speed was 49.6907
```

**Faulty data.** Here are the results of supplying two kinds of invalid data (greeting and closing comments have been omitted):

---

```
// ----- Compute the average speed of your car on a trip.
#include <stdio.h>
int main( void )
{
    int begin_miles, end_miles    // Odometer readings.
    int miles;                   // Total miles traveled.
    double hours, minutes        // Duration of trip.
    double speed;                // Average miles per hour for trip.
    puts( "\n Miles Per Hour Computation \n" );

    printf( " Odometer reading at beginning of trip: " );
    scanf( "%i", &begin_miles );
    while (begin_miles < 0) {
        printf( " Re-enter; odometer reading must be positive: " );
        scanf( "%i", &begin_miles );
    }

    printf( " Odometer reading at end of trip: " );
    scanf( "%i", &end_miles );
    while (end_miles <= begin_miles) {
        printf( " Re-enter; input must be > previous reading: " );
        scanf( "%i", &end_miles );
    }

    printf( " Duration of trip in hours and minutes: " );
    scanf( "%lg%lg", &hours, &minutes );
    hours = hours + (minutes / 60);
    while (hours < 0.0) {
        printf( " Re-enter; hours and minutes must be >= 0.\n" );
        scanf( "%lg%lg", &hours, &minutes );
        hours = hours + (minutes / 60);
    }

    miles = end_miles - begin_miles;
    speed = miles / hours;
    printf( " Average speed was %g \n", speed );

    return 0;
}
```

Figure 3.15. Input validation using while.

1. **Problem scope:** Write a short program for Joe that will compute the price per gallon for one grade of gas, in U.S. funds, that is equivalent to Betty's price for that grade of gas.
  2. **Inputs:** (1) The current exchange rate, in U.S. dollars per Canadian dollar. This rate varies daily. (2) The Canadian prices per liter for one grade of gasoline.
  3. **Constants:** The number of liters in 1 gallon = 3.78544
  4. **Formula:**
- $$\frac{\$_{US}}{\text{gallon}} = \frac{\$_{Canadian}}{\text{liter}} * \frac{\text{liters}}{\text{gallon}} * \frac{\$_{US}}{\$_{Canadian}}$$
5. **Output required:** Echo the inputs and print the equivalent U.S. price.
  6. **Computational requirements:** All the inputs will be real numbers.
  7. **Limitations:** The exchange rate and the price for gas should be positive. If the user enters an incorrect input, an error message should be displayed and another opportunity given to enter correct input.

**Figure 3.16. Problem specification: Gas prices.**

```

Odometer reading at beginning of trip: -1
Re-enter; odometer reading must be positive: 061234
Odometer reading at end of trip: 061521
Duration of trip in hours and minutes: 5 28
Average speed was 52.5

Odometer reading at beginning of trip: 023498
Odometer reading at end of trip: 022222
Re-enter; input must be > previous reading: -32222
Re-enter; input must be > previous reading: 032222
Duration of trip in hours and minutes: 148 43
Average speed was 58.6618

```

## 3.9 An Application

We have analyzed several small programs; now it is time to show how to start with a problem and synthesize a program to solve it. We will create a program for Joe Smith, the owner of a gas station in Niagara Falls, New York. Joe advertises that his prices are cheaper than those of his competitor, Betty, across the border in Niagara Falls, Ontario. To be sure that his claim is true, he computes the U.S. equivalent of Betty's rates daily. Joe's employee reads Betty's pump prices on the way to work each day, and Joe uses his Internet connection to look up the current exchange rate (U.S. dollars per Canadian dollar). Canadian gas is priced in Canadian dollars per liter. U.S. gas is priced in U.S. dollars per gallon. The conversion is too complicated for Joe to do accurately in his head. Figure 3.16 defines the problem and specifies the scope and properties of the desired solution. We will write a solution step by step. As we go along, we will write a comment for every declaration and any part of the code we defer to a later step.

### Step 1. Writing the specification.

Sometimes you will be given a specification, like that in Figure 3.16, and you can begin to plan your strategy based on it. Much of the time, though, you will be given only a general description, as in the previous paragraph. You can fill in many of the details of the specification directly, but you might need to look up in a reference book things like constants or formulas, and you may need to decide the level of accuracy to maintain in your calculations. Until you have completed this step, you will be wasting time by trying to jump into writing the program.

Rate	Can.\$/liter	U.S.\$/gallon
1.0	\$0.26417	\$1.00
1.0	\$1.00	\$3.78544
-0.001	\$1.00	Error
1.0	-\$0.87	Error
0.7412	\$0.55	\$1.543173

Figure 3.17. Test plan: Gas prices.

### Step 2. Creating a test plan.

Before beginning to write the program, we plan how we will test it. The first test case should be something that can be computed in one's head. We note that one of the simplest computations will occur when the exchange rate is 1.0. Then, if the price per liter is the same as the number of gallons per liter, the price per gallon should be \$1.00. We enter this set of numbers as the first line of the test plan in Figure 3.17. We enter the inverse case as the second line of the table: For a price of \$1.00 Canadian per liter, the U.S. price should be the same as the conversion factor for liters per gallon.

As a third test case, we enter an unacceptable conversion rate; we expect to see an error comment in response. We also must test the program's response to an invalid gas price, so we add a line with a negative price. This is called “black-box testing”: the test values are drawn from the specification or from general knowledge of the kinds of data that often cause trouble. They could be chosen by someone with no knowledge of the code itself. (The code could be inside a black box.)

Finally, we enter a typical conversion rate and a typical price per liter, expecting to see an answer that is consistent with real prices. We use a hand calculator to compute the correct answer. We now have five lines in our test plan, which is enough for a simple program that tests for acceptable inputs.

### Step 3. Starting the program.

First, we write the parts that remain the same from program to program, that is, the `#include` command and the first and last lines of `main()` with the greeting message and termination code. The dots in the code represent the unfinished parts of the program that will be filled in by later coding steps.

```
#include <stdio.h>
...
        // Space for #defines.
int main( void )
{
    ...
        // Space for declarations.
    puts( "\n Gas Price Conversion Program \n" );
    ...
        // Input statements.
    ...
        // Computations.
    ...
        // Output statements.
    return 0;
}
```

### Step 4. Reading the data.

The exchange rate is a number with decimal places, so we declare a `double` variable to store it and put the declaration at the top of `main()`:

```
double US_per_Can;    // Exchange rate, $_US / $_Canadian
```

We decide to use a data validation loop to prompt for and read the current exchange rate, so we write down the parts of a `while` validation loop that always are the same, modifying the loop test, prompts, formats, and variable names, as appropriate, for our current application. This code goes into the `main()` program in the second spot marked by the dots.

```
printf( " Enter the exchange rate, $US per $Can: " );
scanf( "%lg", &US_per_Can );
```

```

while (US_per_Can < 0.0) {
    printf( " Re-enter; rate must be positive: " );
    scanf( "%lg", &US_per_Can );
}

```

When writing the calls on `scanf()`, remember to use `%lg` in the format for type `double` and put the ampersand before the variable name.

Next, we must read and validate the Canadian gas price. We declare a variable with a name that reminds us that the input is the price in Canadian dollars for a liter. We also remember to declare a variable for the price in U.S. dollars.

```

double C_liter;           // Canadian dollars per liter
double D_gallon;          // US dollars per gallon

```

Now we write another data validation loop, modifying the loop test, prompts, formats, and variable names, as needed. We write it in the program after the first loop.

```

printf( " Canadian price per liter: " );
scanf( "%lg", &C_liter );
while (C_liter < 0.0) {
    printf( " Re-enter; price must be positive: " );
    scanf( "%lg", &C_liter );
}

```

### Step 5. Converting the gasoline price.

We defined a constant for liters per gallon at the very top of the program. Now we are ready to compute the price per gallon. Remember that we do not use an `=` sign or a semicolon in a `#define` command:

```
#define LITR_GAL 3.78544
```

We check the conversion formula given in the specification, making sure that the units do cancel out and leave us with dollars per gallon. Then we write the code for it and a `printf()` statement to print the answers:

```

D_gallon = C_liter * LITR_GAL * US_per_Can;
printf( "\n Canada: $%g USA: $%g \n", C_liter, D_gallon );

```

### Step 6. Testing the completed program.

The finished program is shown in Figure 3.18. Now we run the program and enter the first data set from the test plan. The results are

```

Gas Price Conversion Program

Enter the exchange rate, $US per $Can: 1.0
Canadian price per liter: .26417

Canada: $0.26417 USA: $1

```

Here is a test run using ordinary data (the last line in the test plan):

```

Gas Price Conversion Program

Enter the exchange rate, $US per $Can: .7412
Canadian price per liter: .55

Canada: $0.55 USA: $1.54317

```

Finally, we run the program twice again to test the error handling (the greeting message has been omitted):

```

Enter the exchange rate, $US per $Can: -0.001
Re-enter; rate must be positive: 1.001
Canadian price per liter: 1.00

Canada: $1 USA: $3.78923

```

```
-----
Enter the exchange rate, $US per $Can: 1.0
Canadian price per liter: -.087
Re-enter; price must be positive: 2.30

Canada: $2.3 USA: $8.70651
```

## 3.10 What You Should Remember

### 3.10.1 Major Concepts

The C language contains many facilities not mentioned yet, and those that have been introduced can be used in many more ways than demonstrated here. It can take years for a programmer to become truly expert in this language. In the face of this complexity, a beginner copes by starting with simple applications, mastering the basic concepts, and learning only the most important details. In this chapter, we have taken a preliminary look at the most basic and important elements of the C language and how they can be combined into a simple but complete program. These are grouped into related areas and summarized.

- **Language elements:**

The C language contains elements (called commands, operators, and functions) that can be translated to

---

```
// -----
// Compute the equivalent prices for Canadian gas and U.S. gas
//
#include <stdio.h>
#define LITR_GAL 3.78544

int main( void )
{
    double US_per_Can;           // Exchange rate, $_US / $_Canadian
    double C_liter;              // Canadian dollars per liter
    double D_gallon;             // US dollars per gallon

    puts( "\n Gas Price Conversion Program \n" );
    printf( " Enter the exchange rate, $US per $Can: " );
    scanf( "%lg", &US_per_Can );
    while (US_per_Can < 0.0) {
        printf( " Re-enter; rate must be positive.\n " );
        scanf( "%lg", &US_per_Can );
    }

    printf( " Canadian price per liter: " );
    scanf( "%lg", &C_liter );
    while (C_liter < 0.0) {
        printf( " Re-enter; price must be positive: " );
        scanf( "%lg", &C_liter );
    }

    D_gallon = C_liter * LITR_GAL * US_per_Can;
    printf( "\n Canada: $%g USA: $%g \n", C_liter, D_gallon );

    return 0;
}
```

---

Figure 3.18. Problem solution: Gas prices.

groups of instructions built into a computer's hardware. It provides names or symbols for actions such as *add* (+) and *compare* (==) but not for complex activities like *solve this equation* or abstract activities such as *think*.

- Overall program structure:
  - An `#include` command is needed at the top of your program to allow your program to use system library functions.
  - A program must have a `main()` function.
  - The `main()` function starts with a series of declarations.
  - A series of statements follows the declarations.
  - A program should start with a statement that prints a greeting and gives instructions for the user.
- Types, objects, and declarations:
  - A program uses the computer's memory to create abstract models of real-world objects such as people, buildings, or numbers. Declarations are used to create and name these objects and may also give them initial values.
  - Declarations are grouped at the top of a program block.
  - Variables and `const` variables are objects; their names are used like nouns in English as the subjects and objects of actions.
  - An object has a name, a location, and a value. The compiler assigns the location for the object. We can give it a value by initializing it in the declaration or by assigning a value to it later. An object that has not been given a value is said to be *uninitialized* or to contain *garbage*.
  - Every object has a type. Types are like adjectives in English: The type of an object describes its properties and how it may be used. The three basic data types seen so far are `int`, `char`, and `double`.
  - A constant object must be initialized in the declaration and its value cannot be changed.
  - A `#define` command can be written at the top of a program to create a symbolic name for a literal constant.
- Simple statements:
  - The programmer combines operations and functions into a series of sentence-like statements that describe the actions to be carried out and the order in which they must happen.
  - Each statement tells the computer what to do next and what variables and constants to use in the process.
  - When a program is executed, the instructions in the program are run in order, from beginning to end. That sequential order can be modified by control instructions that allow for choices and repetition of blocks of statements.
  - The `scanf()` statements perform input. They let a human being communicate with a computer program. If a program requires the user to enter data, the input statement should be preceded by an output statement that displays a user prompt.
  - The `puts()` and `printf()` statements perform output. These statements let a computer program communicate with a human being.
  - An assignment statement can perform a calculation and store the result in a variable so that it can be used later. In general, calculations follow the basic rules of mathematics.
- Compound statements:
  - Statements can be grouped into blocks with braces.
  - The simple `if` statement is a conditional control statement. It has a condition and one block of code that is executed when the condition is true.
  - The `if...else` statement is a conditional control statement that has a condition and two blocks of code, a true clause and a false clause. When an `if...else` statement is executed, the condition is tested first and this determines which block of code is executed.
  - The `while` statement is a looping control statement. It has a condition and one block of code. The condition is tested first, and if the condition is true, the block is executed. Then the condition is retested. Execution and testing are repeated as long as the condition remains true.
  - The counting loops seen so far require that a counter variable be initialized prior to the loop and updated in some manner each time through the loop.

### 3.10.2 Programming Style

- A comment should follow each variable declaration to explain the purpose of the variable.
- Input data should be checked for validity: Garbage input causes garbage output.
- When writing your own programs, it often helps to model your work after a sample program that does a similar task. This makes it easier to find a combination of input, calculation, output, and control statements that are consistent with each other and work gracefully together.
- Indentation is important for readability. A programmer should adopt a meaningful indentation style and follow it consistently.
- Line up the words `if` and `else` with the `}` braces that close each clause in the same column. Indent all the statements in each clause. This assures a neat appearance and helps a reader find the end of the clause.<sup>20</sup>
- If the clause to be executed in one part of an `if` statement is short and the other part is long, put the short clause first. This helps the reader see the whole picture easily. For example, suppose the program must test an input value to determine whether it is in the legal range. If it is legal, several statements will be used to process it. If it is illegal, the program will print an error comment and terminate execution, which takes only two lines of code. This program should be written with the error clause first (immediately following the `if` test) and the normal processing following the `else`.
- Where possible, avoid writing the same statement twice. This makes the program clearer and easier to debug. For example, do not put the same statement in both clauses of an `if` statement; put it before the `if` or after it.
- If both the `true` clause and the `false` clause are single statements, the entire `if` statement can be written on two lines without braces (`{` and `}`) to begin and end the clauses. If either clause is longer than one line, both clauses should be written with braces.

### 3.10.3 Sticky Points and Common Errors

- When you compile a program and get compile-time error comments, look at the first one first. One small error early in the program can produce dozens of error comments; fixing that single error often will make many comments go away.
- If you misspell a word, it becomes a different word in the eyes of the compiler. This is the first thing to check when you do not understand a compile-time or link-time error comment.
- An extra semicolon after the condition in an `if` or `while` statement will end the statement, and the code that should be within the `if` or `while` statement will be outside it. For example, suppose the `while` statement in the countdown program (Figure 3.14) were written incorrectly:

```
m = ITERATIONS;
while (m > 0);
{   printf( " %i.  \n", m );
    m = m - 1;
}
```

The programmer will expect to see 20 lines printed on the page, with the first line numbered 20 and the last numbered 1. Instead, the semicolon ends the loop, which therefore has no body at all. The update line `m = m - 1;` is outside the loop and cannot be reached. The program will become an infinite loop because nothing *within the loop* will decrement the loop variable, `m`.

- A missing semicolon will not be discovered until the compiler begins working on the next line. It will tell you that there is an error, but give the wrong line number. Always check the previous line if you get a puzzling error comment about syntax.

---

<sup>20</sup>This layout scheme is advocated by *Recommended C Style and Coding Standards*, guidelines published in 1994 by experts at Bell Laboratories.

- Quotation marks, braces, and comment-begin and -end marks come in pairs. If the second mark of a pair is omitted, the compiler will interpret all of the program up to the next closing mark as part of the comment or quote. It will produce odd and unpredictable error comments.
- If the output seems to make no sense, the first thing to check is whether the declared type of each variable on the output list matches the conversion specification used to print it. An error anywhere in an output format can affect everything after that on the line. If this does not correct the problem, add more diagnostic printouts to your program to display every data item calculated or read as input. If the input values are correct and the calculated values are wrong, check your formulas for precedence errors and check your function calls for errors.
- If the input values are wrong when you echo them, make sure you have ampersands before the names of the variables in the `scanf()` statement. Check also for type errors in the conversion specifiers in the format.

### 3.10.4 New and Revisited Vocabulary

These important terms and concepts were presented in this chapter:

lexical analysis	precision	assignment (=)
preprocessor command	modifier	expression
program block	identifier	operators (+, -, *, /)
declaration	local name	precedence
statement	undefined value	sequential execution
keyword	garbage	control statement
literal constant	object diagram	condition
symbolic constant	string	loop test
constant variable	prompt	loop body
variable	stream	loop counter
data value	buffer	data validation loop
data type	format	flow diagram
initializer	conversion specifier	black-box testing

The following C keywords and functions were presented in this chapter:

<code>#include</code>	<code>while</code> loop	<code>double</code>
<code>#define</code>	<code>\n</code> (newline character)	<code>int</code>
<code>main()</code>	<code>&amp;</code> (address of)	<code>char</code>
<code>return</code> statement	<code>&lt;stdio.h&gt;</code>	<code>const</code>
{...} (block)	<code>stdin</code>	<code>scanf()</code>
<code>/*...</code> (comment)	<code>stdout</code>	<code>puts()</code>
<code>if...else</code> statement	<code>stderr</code>	<code>printf()</code>

### 3.10.5 Where to Find More Information

- The complete set of standard C keywords is given in Appendix C.
- A full discussion of types `int` and `double` is given in Chapter 7 where we deal with representation, overflow, and the imprecise nature of floating point numbers.
- Additional simple and compound types are introduced in Chapters 10 through 14.
- Storage classes are introduced and used as follows:
  - `const`: Introduced here and used hereafter.
  - `volatile`: Introduced and used in Chapter 13.
  - `auto`: Introduced in Chapter 19, used throughout.
  - `extern`: Introduced in Chapter 19, used in Chapter 20.3.

- **static**: Introduced in Chapter 19, used in Chapter 20.3.
- **register**: Introduced in Chapter 19, not used in a program.
- Many more operators and the details of operator precedence and associativity are given in Chapter 4.
- Program design, with pseudocode and top-down development, is revisited in Chapters 5, 6, and 9.
- Operators associated with pointers are given in Chapter 11, those for bit manipulation are given in Chapter 15, those for accessing a **struct** in chapter 13, and the conditional operator is explained in Appendix D.
- The third basic control structure, functions, will be introduced in Chapter 5
- The other two loop statements will be presented in Chapter 6.
- As other types of data are introduced in Chapters 7 through 11, more details about formats will be presented.
- The complexities of streams will be explored in Chapter 14.

## 3.11 Exercises

### 3.11.1 Self-Test Exercises

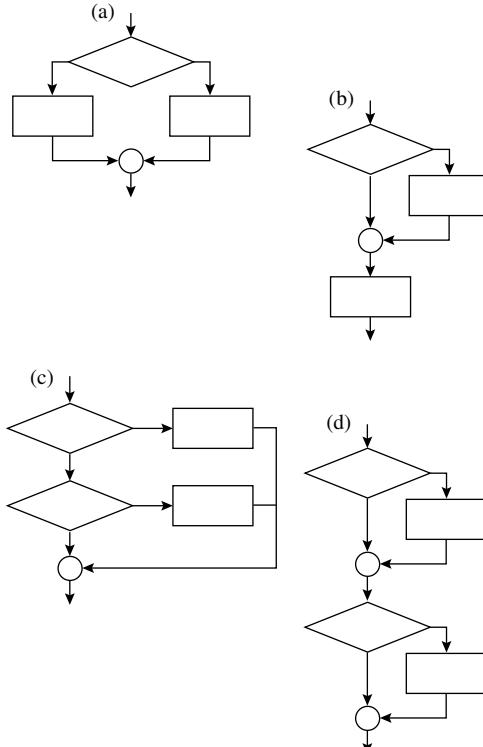
1. Four code fragments and four flow diagrams follow. Note that each diagram has two action boxes and one or two question boxes. All four represent distinct patterns of logic. Match each code fragment to the corresponding flow diagram and show how the code fits into the boxes.

```
(1)
if (radius < 0) {
volume = 0;
}
if (height < 0) {
volume = 0;

(2)
if (radius < 0) {
volume = 0;
}
else if (height < 0) {
volume = 0;
}

(3)
if (t < 1) {
v = t;
}
else {
v = 1;
}

(4)
if (rad > 100)
puts( "Too big" );
area = PI * rad * rad;
```



2. What is wrong with each of the following declarations?

```
(a) int d; a = 5;
(b) doubel h;
(c) int h = 2.5;
```

- (d) `const double g;`  
 (e) `character middle_initial;`  
 (f) `double h = 2.0 * x;`  
 (g) `integer k = 0;`  
 (h) `char gender = "M";`
3. What conversion specifier do you use in an output format for an `int` variable? For a `double` variable? For a character?
4. Find the error in each of the following declarations.
- (a) `integer count;`  
 (b) `real weight ;`  
 (c) `int k; count;`  
 (d) `character gender;`  
 (e) `duble age;`
5. What happens when you omit the ampersand (address of) before a variable name in a `scanf()` statement? To find out, delete an ampersand in one of the sample programs. Try to compile and run the resulting program.
6. What happens when you type an ampersand before a variable name in a `printf()` statement? Try it.
7. What happens when you type a comment-begin mark but forget to type (or mistype) the matching comment-end mark? To find out, delete a comment-end mark in one of the sample programs and try to compile the result.
8. What happens when you type a semicolon after the closing parenthesis in a simple `if` statement? Try it.
9. What is wrong with each of the following `if` statements? They are supposed to identify and print out the middle value of three `double` values, `x`, `y`, and `z`.
- (a) `if (x < y < z) printf( "y=%g", y );  
 else if (y < x < z) printf( "x=%g", x );  
 else printf( "z=%g", z );`
- (b) `if (x < y)  
 if (y < z) printf( "y=%g", y );  
 if (z < y) printf( "z=%g", z );  
else  
 if (x < z) printf( "x=%g", x );  
 if (z < x) printf( "z=%g", z );`
- (c) `if (x > y)  
{ if (x < z);  
 printf( "x=%g", x );  
 else printf( "z=%g", z );  
}  
else  
{ if (y < z);  
 printf( "y=%g", y );  
 else printf( "z=%g", z );  
}`

### 3.11.2 Using Pencil and Paper

1. What does your compiler do when you misspell a keyword such as `while` or `else`? What happens when you misspell a function name such as `main()` or `scanf()`? Try it.

2. What happens when you type a semicolon after the closing parenthesis in a `while` statement? Try it.
3. What conversion specifier do you use in an input format for an `int` variable? For a `double` variable? For a character variable?
4. Find the error in each of the following preprocessor commands.

(a) `#include <stdio>`  
 (b) `#define NORMAL = 98.6`  
 (c) `#include stdio.h`  
 (d) `#define TOP 1,000`  
 (e) `#define LOOPS 10;`  
 (f) `#include <studio.h>`

5. Given the declarations on the first three lines that follow, find the error in each of the following statements:

```
int age, count;
double price, weight;
char gender;
```

(a) `scanf( "%g", &price );`  
 (b) `scanf( "%c", &gender );`  
 (c) `scanf( "%d", &weight );`  
 (d) `printf( "%i", &count );`  
 (e) `printf( "%lg", price );`  
 (f) `printf( "%c", gendre );`

6. Draw a flow diagram of the following program and use your diagram to trace its execution. What is the output?

```
#include <stdio.h>
int main( void )
{
    int k, m;
    k = 0;
    m = 1;
    while (k <= 3) {
        k = k + 1;
        m = m * 3;
    }
    printf( "k = %i m = %i \n", k, m );
    return 0;
}
```

7. In the following program, circle each error and show how to correct it:

```
#include (stdio.h)
int main (void)
{
    integer k;
    double x, y, z
    printf( Enter an integer: );
    scanf( "%i", k );
    printf( "Enter a double: ");
    scanf( "%g", &x );
    printf( "Enter two integers: ");
    scanf( "%lg", &y, &z );
    printf( "k= %i X= %g \n", &k, &x );
    printf( "y= %i z= %g \n" y, z );
}
```

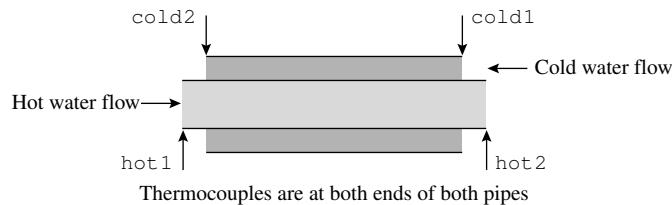
8. Draw a flow diagram for each set of statements, then trace their execution and show the output. Use these values for the variables:  $w=3$ ,  $x=1$ ,  $y=2$ ,  $z=0$ .

```
(a) if (x < y) z=1; if (x > w) z=2;
   printf(" %i\n", z);
(b) if (x < y) z=1; else if (x > w) z=2;
   printf(" %i\n", z);
(c) if (w < x) z=1; else if (w > y) z=2; else z=3;
   printf(" %i \n", z);
```

### 3.11.3 Using the Computer

1. Heat transfer.

Complete the following program for heat transfer that already has been designed and partially coded. The program analyzes the results from a heat transfer experiment in which hot water flows through a pipe and cold water flows in the opposite direction through a surrounding, concentric pipe. Thermocouples are placed at the beginning and end of both pipes to measure the temperature of the water coming in and going out, as diagrammed here.



This is a standard engineering calculation. To calculate the heat transfer, we need the average hot temperature, the average cold temperature, and the change in temperature from input to output for both hot and cold pipes.

Your job is to start with the incomplete program in Figure 3.19, fill in the boxes according to the instructions that follow in each box, and make the program work. Use the following test plan.

Test Plan	Data				Answers			
	Inlets		Outlets		Means		Differences	
Test Objective	Hot	Cold	Hot	Cold	Hot	Cold	Hot	Cold
Easy to calculate	100	0	50	50	75	25	-50	50
Another legal input	120	35	100	50	110	42.5	-20	15

- (a) Get a copy of the file `flex_heat.c`, which contains the partially completed program in Figure 3.19. Write program statements to implement the actions described in each of the comment boxes. Delete the existing comments and add informative ones of your own.
- (b) Compile your code and test it using the test plan given above. Prepare proper documentation for the project including the source code, output from running the program on the test data, and hand calculations that prove the output is correct.
2. Temperature Conversion.

Complete a program for temperature conversion that has been designed and partially coded. The program will read in a temperature in degrees Fahrenheit, convert it into the equivalent temperature in degrees Celsius, and display this result. Your job is to start with the incomplete program in Figure 3.20 fill in the boxes according to the instructions given below in each box, and make the program work.

- (a) Make an appropriate test plan for your program following the layout scheme below.

Test objective	Input Fahrenheit	Output Celsius
Easy-to-calculate input		
Another legal input		
Minimum legal input		
Out-of-range input		

- (b) Start coding with a copy of the file `flex_temp.c`, which contains the partially completed program in Figure 3.20. Following the instructions given in each comment box, write program statements to implement the actions described. Delete the existing comments and add informative ones of your own. Compile and run your code, testing it according to the first line of your plan. Check the answer. If it is correct, print it out and go on to the rest of your test plan. If it is incorrect, fix it

```
// This program will compute heat characteristics of a concentric-pipe
// heat exchanger, including mean hot and cold temperatures, and the
// differences in temperatures of the ends of the pipes.

#include <stdio.h>
int main( void )
{
    double hot1;           // hot inlet temperature
    double hot2;           // hot outlet temperature
    double cold1;          // cold inlet temperature
    double cold2;          // cold outlet temperature
    double mean_hot;        // average of hot temperatures
    double mean_cold;       // average of cold temperatures
    double dthot;           // difference in hot temperatures
    double dtcold;          // difference in cold temperatures

    puts( "Heat Transfer Experiment" );

Prompt user to enter four inlet and outlet temperatures.
      Use a separate prompt for each temperature you read.
      Use scanf() to read each and store in the appropriate variable.



Calculate the mean of the hot temperatures (their sum divided by 2)
      and the mean of the cold temperatures. Save each value in an
      appropriate variable.
      Calculate dthot (difference in hot temperatures = the hot outlet
      temperature minus the hot inlet temperature) and dtcold.
      Again save these in the appropriate variables.
      Put comments on these lines describing the calculations.



Write printf() statements to echo the four input temperatures.
      Write printf() statements to print the four calculated numbers.
      Label each output clearly, so we can tell what it means.
      Use \n and spaces in your formats to make the output easy to read.


    return 0;
}
```

Figure 3.19. Sketch for the heat transfer program.

```

// -----
// This program will convert temperatures from degrees Fahrenheit into
// degrees Celsius. Input temperature must be above absolute zero.

#include <stdio.h>

Define a constant for absolute zero in Fahrenheit (-459.67).

int main( void )
{
    double fahr;          // Temperature in Fahrenheit degrees
    double cels;          // Temperature in Celsius degrees

    puts( "\n Temperature Conversion Program" );

    Prompt the user to enter a temperature in degrees Fahrenheit.
    Use scanf() to read it and store it in the appropriate variable.
    Write a printf() statement that will echo the input.

    Test whether the input temperature is less than absolute zero.
    If so, print an error comment.
    If not, calculate the temperature in degrees Celsius according
    to the formula  $C = \frac{5.0}{9.0} * (F - 32.0)$  and print the answer.

    Label the output clearly.
    Use newlines and spaces in your formats to make the output
    easy to read.

    return 0;
}

```

**Figure 3.20.** Sketch for the temperature conversion program.

and test it again until it is correct. When the answers are correct, print out the program and its output on these tests, and hand them in with the test plan.

### 3. Sales Tax.

The text website contains a complete program to solve the problem described below. However, the program will not compile and may have logical errors. Download the program and debug it, according to your specification. Hand in the debugged program and its output.

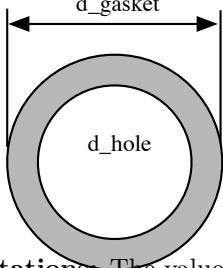
Given the cost of a purchase, and a character 'T' for taxable or 'N' for non-taxable, compute the sales tax and total price. Let the tax rate be 6%.

### 4. Gasket area.

A specification for a program that computes the area of a ring gasket is given in Figure 3.21. From this, the flow diagram in Figure 3.22 was constructed.

- Develop a test plan for this program based on the problem specification in Figure 3.21.
- Write a program based on the algorithm in Figure 3.22.
- Make sure all of your prompts and output labels are clear and easy to read.
- Test your program using the test plan you devised.
- Turn in the source code and output results from your test.

- (a) **Problem scope:** Write a program that will calculate the surface area of a ring gasket.
- (b) **Inputs:** The outer diameter of the gasket, `d_gasket`, and the diameter of the hole in the center of the gasket, `d_hole`. These should be given in centimeters.



**Formula:**

$$\text{area} = \frac{\pi \times (d_{\text{gasket}}^2 - d_{\text{hole}}^2)}{4}$$

- (c) **Limitations:** The value of `d_gasket` must be nonnegative. The ratio of `d_hole` to `d_gasket` must be greater than 0.3 and less than 0.9.
- (d) **Constant:**  $\pi = 3.14159265$
- (e) **Output required:** The surface area of the gasket. (Echo the inputs also.)
- (f) **Computational requirements:** All the inputs will be real numbers.

Figure 3.21. Problem specification: Gasket area.

#### 5. Skyscraper.

The text website contains a complete program to solve the problem described below. However, the specification and test plan are missing and the program will not compile and may have logical errors. Write a specification and test plan, then download the program and debug it, according to your specification. Hand in your work plus the debugged program and its output.

Assume that the ground floor of a skyscraper has 12-foot ceilings, while other floors of the building have 8-foot ceilings. Also, the thickness in between every floor is 3 feet. On top of the building is a 20-foot flagpole. If the building has  $N$  stories altogether, and  $N$  is given as an input to your program, calculate the height of a blinking red light at the top of the flagpole and print out this height.

#### 6. Weight conversion.

Write a program that will read a weight in pounds and convert it to grams. Print both the original weight and the converted value. There are 454 grams in a pound. Design and carry out a test plan for this program.

#### 7. Distance conversion.

Convert a distance from miles to kilometers. There are 5,280 feet per mile, 12 inches per foot, 2.54 centimeters per inch, and 100,000 centimeters per kilometer.

#### 8. More Temperature Conversion.

Revise the specification, test plan and program from problem 4 so that the user can enter a temperature in either Celsius or Fahrenheit, and the program will convert it to the other system. The inputs should be a temperature and a character, "C" for Celsius or "F" for Fahrenheit. Use an `if` statement to test which letter was entered, then perform the appropriate conversion. Echo the input and label the output properly. The formula for conversion from Celsius to Fahrenheit is

$$\text{Fahrenheit} = \text{Celsius} \times \frac{9}{5} + 32$$

#### 9. Meaningful change.

Write a program that will input the cost of an item from the user and output the amount of change due if the customer pays for the purchase with a \$20 bill. What kinds of problems might this program have? (Think about unexpected inputs.) Design a test plan to detect these problems. Use an `if` statement to validate your data to prevent meaningless outputs.

#### 10. A snow job.

Your snow blower clears a swath 2 feet wide. Given the length and width of your rectangular driveway, calculate how many feet you will need to walk to clear away all the snow. Be sure to include the steps you take when you turn the snow blower around. Write a program that contains validation loops for the two input dimensions, echoes the input, and prints out the calculated distance.

11. Pluses and minuses.

Your program will be used as part of a quality control system in a factory that makes metal rods of precise lengths. The current batch of rods is supposed to be 10 cm long. An automated measuring device measures the length of each rod as it comes off the production line. These measurements are automatically sent to a computer as input. Some rods are slightly shorter and some slightly longer than the target length. Your program must read the measurements (use `scanf()`). If the rod is too short, add it to a `short_total`; otherwise, add it to a `long_total`. Also, count it with either the `short_counter` or the `long_counter`. After each batch of 20 rods, print out your totals, counters, and the average length of the rods in the batch.

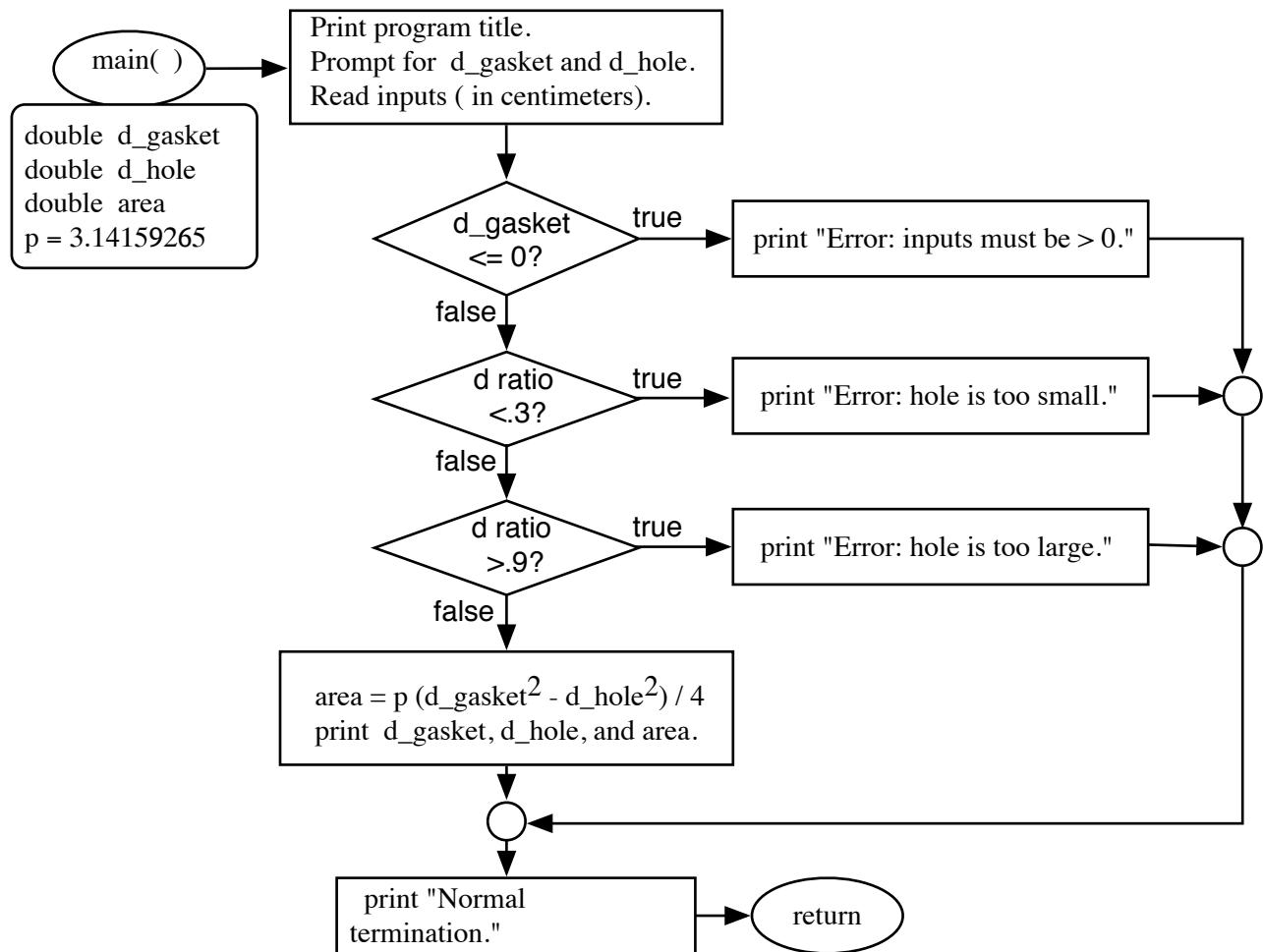


Figure 3.22. Flow diagram for the gasket area program.

# Part II

# Computation



# Chapter 4

## Expressions

Computation is central to computer programming. We discuss how the usual mathematical operations can be applied to variables and introduce the concepts of precedence and associativity that govern the meaning of an expression in both mathematical notation and in C. Parse trees are introduced to help explain the structure and meaning of expressions.

Finally, declarations, expressions, and parse trees are brought together in a discussion of program design and testing methodology. We propose a problem, develop a program for it, analyze how we should test the program, and show how to use data type rules and parse trees to find and correct an error in the coding.

### 4.1 Expressions and Parse Trees

An **expression** is like an entire sentence in English. It specifies how verbs (operators and functions) should be applied to nouns (variables and constants) to produce a result. In this section, we consider the rules for building expressions out of names, operators, and grouping symbols and how to interpret the meanings of those expressions. We use a kind of diagram called a *parse tree* to see the structure of an expression and to help us understand its meaning.

#### 4.1.1 Operators and Arity

C has several classes of operators in addition to those that perform arithmetic on numbers. These include comparison and logical operators, bit-manipulation operators, and a variety of operators that are found in C but not in other languages. Each group of operators is intended for use on a particular type of object. Many will be introduced in this chapter; others will be discussed in detail in the chapters that introduce the relevant data types. For example, operators that work specifically with integers are explained in Chapter 7. Operators that deal with pointers, arrays, and structured types are left for later chapters. Issues to be considered with each class of operator include the type of data on which it operates, its precedence and associativity, any unusual rules for evaluation order, and any unexpected aspects of the meaning of an operator.

An **operand** is an expression whose value is an input to an operation. *Operators* are classed as unary, binary, or ternary according to the number of operands each takes. A **binary operator** has two operands and is written between those operands. For example, in the expression  $(z/4)$ , the / sign is a binary operator and z and 4 are its operands. We also say that a binary operator has arity = 2. The **arity** of an operator is the number of operands it requires.

A unary operator (arity = 1), such as - (negate), has one operand. Most unary operators are **prefix operators**; that is, they are written before their operand. Two unary operators, though, also may be written after the operand, as **postfix operators**. There is only one ternary operator (arity = 3), the conditional expression. The two parts of this operator are written between its three operands.

### 4.1.2 Precedence and Parentheses

Many expressions contain more than one operator. In such expressions, we need to know which operands go with which operators. For example, if we write

$$13/5 + 2$$

it is important to know whether this means

$$(13/5) + 2 \quad \text{or} \quad 13/(5 + 2)$$

because the two expressions have different answers.

**Using parentheses.** Parentheses are neither operators nor operands; called *grouping symbols*, they can be used to control which operands are associated with each operator. Parentheses can be used to specify whatever meaning we want; a fully parenthesized expression has only one interpretation. If we do not write parentheses, C uses a default interpretation based on the rules for precedence and associativity. This default is one of the most common sources for errors in writing expressions.

**The precedence table and the default rules.** Experienced programmers use parentheses only to emphasize the meaning of an expression or achieve a meaning different from the default one. They find it a great convenience to be able to omit parentheses most of the time. Further, using parentheses selectively makes expressions more readable. However, if you use the default precedence, anyone reading or writing your program must be familiar with the rules to understand the meaning of the code.

The precedence and associativity of each operator are defined by the C precedence table, given in Appendix B. The rules in this table describe the meaning of any unparenthesized part of an expression. **Precedence** defines the grouping order when different operators are involved, and **associativity** defines the order of grouping among adjacent operators that have the same precedence. The concepts and rules for precedence and associativity are simple and mechanical. They can be mastered even if you do not understand the meaning or use of a particular operator. The portion of the **precedence table** that covers the arithmetic operators is repeated in Figure 4.1, with a column of examples added on the right. We use it in the following discussion to illustrate the general principles.

**Precedence.** In the precedence table, the C operators are listed in order of precedence; the ones at the top are said to have *high precedence*; those at the bottom have *low precedence*.<sup>1</sup> As a convenience, the precedence classes in C also have been numbered. There are 17 different classes. (The higher the number, the higher is the precedence.) Operators that have the same precedence number are said to have *equal precedence*. You will want to mark Appendix B so that you can refer to the precedence table easily; until all the operators are familiar, you will need to consult it often.

**Associativity.** The rule for associativity governs consecutive operators of equal precedence, such as those in the expression  $(3 * z / 10)$ . All these operators with the same precedence will have the same associativity, which is either left to right or right to left.<sup>2</sup> With left-to-right associativity, the leftmost operator is parsed before the ones to its right. (The process of parsing is explained next.) With right-to-left associativity, the rightmost operator is parsed first. In the expression  $(3 / z * 10 \% 3)$ , the three operators have the same precedence and all have left-to-right associativity. We therefore parse the  $/$  first and the  $*$  second, giving this result:  $((3 / z) * 10 \% 3)$ . The parse of this expression is diagrammed in Figure 4.3.

Almost all the unary operators have the same precedence (15) and are written before the operand (that is, in prefix position). The chart shows that they associate right to left. Therefore, in the expression  $(- - x)$ , the second negation operation is parsed first and the leftmost negation operation second, giving this result:  $(- (- x))$ . Restated simply, this rule states “the prefix operator written closest to the operand is parsed first.”

---

<sup>1</sup>Most of the C operators will be described in this chapter; others will be explained in later chapters.

<sup>2</sup>The term *left-to-right associativity* is often shortened to *left associativity* and *right-to-left to right*.

Arity	Symbol	Meaning	Precedence	Associativity	Examples
Unary	-	Negation	15	right to left	-1, -temp
	+	No action		"	+1, +x
Binary	*	Multiplication	13	left to right	3 * x
	/	Division		"	x / 3.0
	%	Integer remainder (modulus)		"	k % 3
	+	Addition		"	x + 1, x + y
	-	Subtraction		"	x - 1, 3 - y

Figure 4.1. Arithmetic operators.

### 4.1.3 Parsing and Parse Trees

*Parsing* is the process of analyzing the structure of a statement. The compiler does this when it translates the code, and human beings do it when reading the code. One way to parse an expression is to write parentheses where the precedence and associativity rules would place them by default. This process is easy enough to carry out but can become visually confusing.

An easier way to parse is to draw a *parse tree*. The **parse tree** shows the structure of the expression and is closely related to what the C compiler does when it translates the expression. A parse tree helps us visualize the structure of an expression. It also can be used to understand the order in which operators will be executed by C, which, unfortunately, is not the same as their order of precedence.<sup>3</sup>

To parse an operator, either write parentheses around it and its operands or draw a tree bracket with one branch under each operand and the stem under the operator. A simple, two-pronged bracket is used for most binary operators. The bracketed or parenthesized unit becomes a single operand for the next operator, as shown in Figures 4.2 and 4.3. Brackets or parenthesized units never collide or cross each other. Figure 4.2 shows a parse tree that uses the precedence rules, and Figure 4.3 shows an expression where all operators have equal precedence and the rule for associativity is used. The steps in parsing an arbitrary expression are these:

1. Write the expression at the top, leaving some space between each operand and operator.
2. Parse parenthesized subexpressions fully before looking at the surrounding expression.
3. If there are postfix unary operators (-- or ++), do them first. These are mentioned here for completeness and will be explained later in the chapter.
4. Next, find the prefix unary operators, all of which associate right to left. Start at the rightmost and bracket each one with the following operand, drawing a stem under the operator. (An operand may be a variable, a literal, or a previously parsed unit.)

<sup>3</sup>Complications of evaluation order are covered in Sections 4.3 and 4.6.3 and Appendix D.

The \* operator has highest precedence so it was parsed first and it “captured” the middle operand. The assignment has lowest precedence so it was parsed last. The arrowhead on the assignment operator bracket indicates that the value of x is updated with the right operand value.

$$\begin{aligned} &x = m + k * y \\ &x = m + (k * y) \\ &x = (m + (k * y)) \\ &(x = (m + (k * y))) \end{aligned}$$

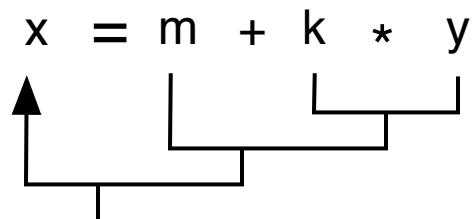
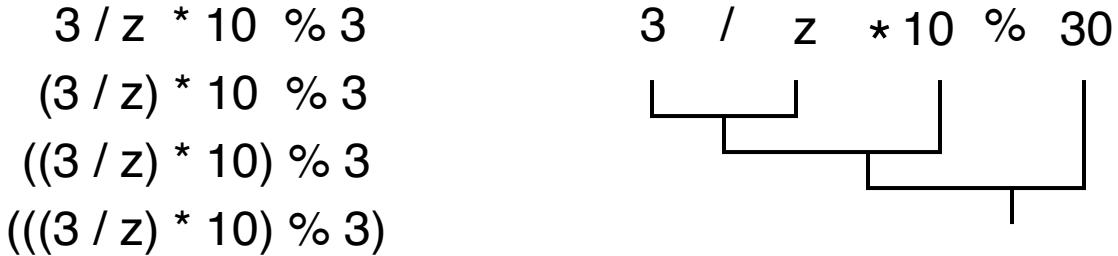


Figure 4.2. Applying precedence.

In the diagram, three brackets are used to show which operands go with each of the operators in the expression  $(3 / z * 10 \% 3)$ . These operators have equal precedence, so they are parsed according to the associativity rule for this group of operators, which is left to right.



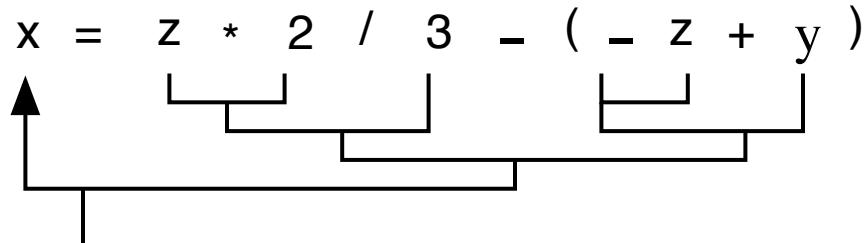
**Figure 4.3. Applying associativity.**

5. Now look at the unparsed operators in the expression and look up the precedence of each one in the table. Parse the operators of highest precedence first, grouping each with its neighboring operands. Then go on to those of lower precedence. For neighboring operators of the same precedence, find their associativity in the table. Proceeding in the direction specified by associativity, bracket each operator with the two adjacent operands.
6. Repeat this process until you finish all of the operators.

Figure 4.4 shows an example of parsing an expression where all the rules are applied: grouping, associativity, and precedence.

**Notes on Figure 4.4. Applying the parsing rules.** The steps in drawing this parse tree are as follows:

1. Subexpressions in parentheses must be parsed first. Within the parentheses, the prefix operator  $-$  is parsed first. Then the lower precedence operator,  $+$ , uses this result as its left operand to complete the parsing in the parentheses.
2. The  $*$  and  $/$  are the operators with the next highest precedence; we parse them left to right, according to their associativity rule.
3. The  $-$  is next; its left operand is the result of  $/$  and its right operand is the result of the parentheses grouping.
4. The  $=$  is last; it stores the answer into  $x$  and returns the value as the result of the expression (this is explained in the next section).



**Figure 4.4. Applying the parsing rules.**

All these operators associate right to left. The left operand of each must be a variable or storage location. The parse diagram for a combination operator is shown in Figure 4.6

Symbol	Example	Meaning	Precedence
=	x = 2.5	Store value in variable	2
+=	x += 3.1	Same as (x = x + 3.1); add value to variable and store back into variable	2
-=	x -= 1.5	Same as (x = x - 1.5); subtract value from variable and store back into variable	2
*=	x *= 5	Same as (x = x * 5); multiply and store back	2
/=	x /= 2	Same as (x = x / 2); divide by and store back	2

Figure 4.5. Assignment and arithmetic combinations.

## 4.2 Arithmetic, Assignment, and Combination Operators

The **arithmetic operators** supported by C are listed in Figure 4.1. All these operators can be used on any pair of numeric operands. The **assignment operator** was described and its uses discussed in Chapter 3. These operators can be combined to give a shorthand notation for doing both actions. For example, the expression ( $k += 3.5$ ) has the same meaning as ( $k = k + 3.5$ ). It means “fetch the current value of  $k$ , add 3.5, and store the result back into  $k$ .” The symbol for a **combination operator** is formed by writing the desired operator symbol followed by an assignment sign, as in  $+=$  or  $*=$ . The arithmetic combinations are listed in Figure 4.5, and a parse tree for assignment combinations in Figure 4.6. All combination operators have the same very low precedence, which means that other operators in the expression will be parsed first. If more than one assignment or combination operator is in an expression, these operators will be parsed and executed right to left.

**Side effects.** The assignment combinations, along with ordinary assignment and the increment and decrement operators (described in the next section), are different from all other operators: They have the **side effect** of changing the value of a memory variable. Assignment discards the old value and replaces it with a new value. The combination operators perform a calculation using the value of a memory variable then store the answer back into the variable. When using a combination operator, the right operand may be a variable

This is the parse tree and evaluation for the expression  $k += m$ . Note that the combined operator is diagrammed with two brackets. The upper bracket is for  $+$ , which adds the initial value of  $k$  to the value of  $m$ . The lower bracket shows that the sum is stored back into  $k$ .

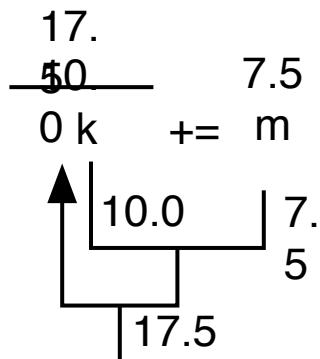


Figure 4.6. Parse tree for an assignment combination.

Each time we halve a number, we eliminate one binary digit. If we halve it repeatedly until it reaches 0 and count the iterations, we know how many binary digits are in the number.

```
#include <stdio.h>
int main( void )
{
    int n;           // Input - the number to analyze.
    puts( "\n Halving and Counting\n" );
    printf( " Enter an integer: " );
    scanf( "%i", &n );
    printf( " Your number is %i, ", n );

    int k = 0;       // Initialize the counter before the loop.
    while (n > 0) { // Eliminate one binary digit each time around loop.
        n /= 2;      // Divide n in half and discard the remainder.

        k += 1;      // Count the number of times you did this.
    }

    printf( " it has %i bits when written in binary. \n\n", k );
    return 0;
}
```

Output:

```
Halving and Counting

Enter an integer: 37
Your number is 37, it has 6 bits when written in binary.
```

**Figure 4.7. Arithmetic combinations.**

or an expression, but the left operand must be a variable, since the answer will be stored in it.

**Using combination operators.** Figure 4.7 shows how two of the arithmetic combinations can be used in a loop. It is a preliminary version of an algorithm for expressing a number in any selected number base; the complete version is shown in Figure 4.23.

#### Notes on Figure 4.7. Arithmetic combinations.

##### *Outer box.*

This loop is slightly different from the counting loops and validation loops seen so far. It simply continues a process that decreases `n` until the terminal condition, `n` equals 0, occurs. More loop types are discussed in Chapter 6.

##### *First inner box: halving the number.*

The statement `n /= 2;` divides `n` by 2, throwing away the remainder. The quotient is an integer (with no fractional part) and is stored back into `n`. Therefore, the value of  $1/2$  is 0, and the value of  $7/2$  is 3. Integer arithmetic is discussed more fully in Chapter 7.

##### *Second inner box: incrementing the counter.*

The operator `+=` is often used to increment loop counters, especially when counting by twos (or any increment not equal to 1). As seen in the next section, the operator `++` is more popular for adding 1 to a counter.

The increment and decrement operators cause side effects; that is, they change values in memory. The single operand of these operators must be a variable or storage location. The prefix operators both associate right to left; the postfix operators associate left to right.

Fixity	Symbol	Example	Meaning	Precedence
Postfix	++	j++	Use the operand's value in the expression, then add 1 to the variable	16
	--	j--	Use the operand's value in the expression, then subtract 1 from the variable	16
Prefix	++	++j	Add 1 to the variable then use the new value in the expression	15
	--	--j	Subtract 1 from the variable then use the new value in the expression	15

Figure 4.8. Increment and decrement operators.

## 4.3 Increment and Decrement Operators

C contains four operators that are not present in most other languages because of the problems they cause, but they are very popular with C programmers because they are so convenient. These are the pre- and postincrement (++) and pre- and postdecrement (--) operators, which let us add or subtract 1 from a variable with only a few keystrokes. The increment operator most often is used in loops, to add 1 to the loop counter. Decrement sometimes is used to count backward. Both normally are used with integer operands, but they also work with variables of type `double`.

The same symbols, ++ and --, are used for both pre- and postincrement, but the former is written before the operand and the latter after the operand. The actions they stand for are the same, too, except that these actions are executed in a different order. Collectively, we call this group the *increment operators*; they are listed in Figure 4.8.

### 4.3.1 Parsing Increment and Decrement Operators

The increment operators have two properties that, when put together, make them unique from the operators considered earlier. First, they are unary operators, and second, they modify memory. The parse diagrams for these operators reflect these differences.

To diagram an increment or decrement operator, bracket it with its single operand (which must be a variable or a reference to a variable), drawing a stem under the operator and an assignment arrowhead to show that the variable will receive a new value. That value also is passed on, down the parse tree, and can be used later in the expression. See Figure 4.9 for examples.

When postincrement or postdecrement is used, it is also possible to have prefix unary operators applied to the same operand. In this case, the postfix operator is parsed first (it has higher precedence), then the prefix operators are done right to left. Thus, the expression `(- x ++)` parses to `(- (x ++))`<sup>4</sup>.

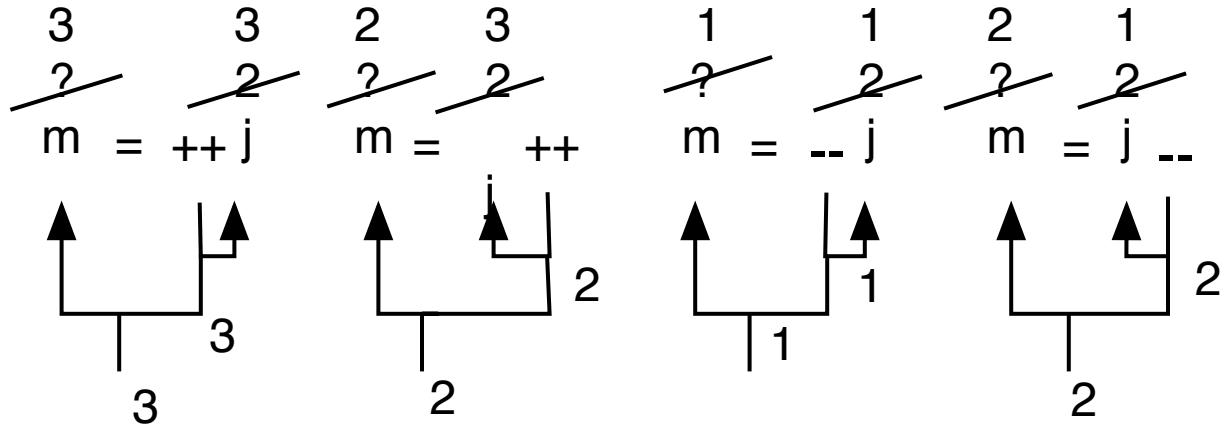
### 4.3.2 Prefix versus Postfix Operators

For simplicity, we will explain the differences between pre- and postfix operators in terms of the increment operators, but everything is equally true of decrement operators; simply change “add 1” to “subtract 1” in the explanations.

Both *prefix* and *postfix increment* operators add 1 to a variable and return the value of the variable for further use in an expression. For example, `k++` and `++k` both increase the value of `k` by 1. If a prefix increment or a postfix increment operator is the only operator in an expression, the results will be the same. However, if an increment operation is embedded in a larger expression, the result of the larger expression will be different for prefix increment and postfix increment operators. Both kinds of increment operator return the value of `k`, to be used in the larger expression. However, the prefix form increments the variable *before* it returns the value, so that the value in memory and the one used in the expression are the same. In contrast, postfix increment returns the original, unincremented value of the variable to the larger expression and increments the value

<sup>4</sup>The Applied C website, Program 4.A, shows a typical use of preincrement in a counting loop.

The parse trees are shown for each of these operators. The arrowheads indicate that a value is stored back into memory.



**Figure 4.9. Increment and decrement trees.**

in memory *afterward*. Thus, the value used in the surrounding expression is 1 smaller than the value left in memory and 1 smaller than the value used in the prefix increment expression.

A further complication with postfix increment is that the change in memory does not have to happen right away. The compiler is permitted to postpone the store operation for a while, and many do postpone it in order to generate more efficient code.<sup>5</sup> This makes postfix increment and decrement somewhat tricky to use properly. However, you can depend on two things: First, if you execute `x++` three times, the value of `x` will be 3 greater than when you started. Second, by the time evaluation reaches the semicolon at the end of the statement, all incrementing and decrementing actions will be complete.

### 4.3.3 Mixing increment or decrement with other operators.

An increment operator is an easy, efficient way to add 1 to the value of a memory variable. Most frequently, increment and decrement operators are used in isolation. However, both also can be used in the middle of an expression because both return the value of the variable for use in further computation.

New programmers often write one line of code per idea. For example, consider the summation loop below. The first line of the loop body increments `x` and the second line uses the new value:

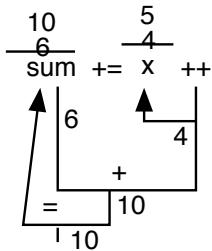
```
x = 1;           // Sum x for x = 1 to N
while (x <= N) { // Leave loop when x exceeds N.
    sum += x;     // Same as sum = sum + x
    ++x;          // Add 1 to x to prepare for next iteration.
}
```

// Go back to the while test.

An advanced programmer is more likely to write the following version, which increments `x` and uses it in the same line. The parse tree and evaluation of the assignment expression are shown in Figure 4.10, for the fourth time through the summing loop.

```
x = 1;           // Sum x for x = 1 to N
while (x <= N) sum += x++;
```

<sup>5</sup>The exact rules for this postponement are complex. To explain them, one must first define sequence points and how they are used during evaluation. This explanation is beyond the scope of the book.



**Figure 4.10.** Evaluating an increment expression.

As you can see, mixing an increment with other operators in an expression makes the code “denser”—more actions happen per line—and it often permits us to shorten the code. By using two side-effect operators, we have reduced the entire loop to one line. However, this happens at the expense of some clarity and, for beginners, at the risk of writing something unintended.

When using an increment or decrement in an expression, the difference between the prefix and postfix forms is crucial. If you use the postfix increment operator, the value used in the rest of the expression will be one smaller than if you use the prefix increment. For example, the loop shown previously sums the fractions  $1/1\dots 1/N$ . If a postfix increment were used instead of the prefix increment, it would try to sum the fractions  $1/0\dots 1/(N-1)$  instead. Of course, dividing by 0 is a serious error that causes immediate program termination on some systems and meaningless results on others. Therefore, think carefully about whether to use a prefix or postfix operator to avoid using the wrong value in the expression or leaving the wrong value in memory. When these operators are used in isolation, such problems do not occur.

**Guidance.** Because of the complications with side effects, increment and decrement operators can be tricky to use when embedded in a complex expression. They are used most often in isolation to change the value of a counter in a loop, as in Figure 4.23. The following are some guidelines for their use that will help beginners avoid problems:

1. Do not mix increment or decrement operators with other operators in an expression until you are an experienced programmer.
2. Do not ever use increment on the same variable twice in an expression. The results are unpredictable and may vary from compiler to compiler or even from program to program translated by the same compiler.
3. Until you are experienced and are sure of the semantics of postfix operators, use prefix increment and decrement operators instead. They are less confusing than postfix increment and decrement operators and cause less trouble for beginners.

## 4.4 The sizeof operator.

When we declare an object, we say what type it will be. Types (sometimes called **data types**) are like adjectives in English: The type of an object describes its size (number of bytes in memory) and how it may be used. It tells the compiler how much storage to allocate for it and whether to use integer operations, floating-point operations, or some other kind of operations on it.

Many types are built into the C language standard; each has its own computational properties and memory requirements. Each type has different advantages and drawbacks, which will be examined in the next few chapters. Later, we will see how to define new types to describe objects that are more complex than simple letters and numbers.

The current popularity of C is based largely on its power and portability. However, the same data type can be different sizes on different machines, which adversely affects portability. For example, the type `int` commonly is two bytes on small personal computers and four bytes on larger machines. Also, some computer memories are built out of words not bytes.

---

This short program demonstrates the proper syntax for using `sizeof`. Note that the parentheses are not needed for variables but they *are* needed for type names.

```
// -----
// Demonstrate the use of sizeof to determine memory usage of data types

#include <stdio.h>
#define e 2.718281828459045235360287471353 // Mathematical constant e.

int main( void ) // how to use sizeof
{
    int s_double, s_int, s_char, s_unknown, k;

    s_double = sizeof (double) ; // use parentheses with a type name
    s_int = sizeof k ; // no parentheses needed with a variable
    s_char = sizeof '%';

    printf( " sizeof double = %i \n sizeof int = %i \n sizeof char = %i \n",
    s_double, s_int, s_char );
    s_unknown = sizeof e ;
    printf( " sizeof e = %i \n", s_unknown );
    return 0;
}
```

Results when run on our workstation:

```
sizeof double = 8
sizeof int = 4
sizeof char = 1
sizeof e = 8
```

---

**Figure 4.11. The size of things.**

The actual size of a variable, in bytes, becomes very important when you take a program debugged on one kind of system and try to use it on another; the variability in the size of a type can cause the program to fail. To address this problem, C provides the `sizeof` operator, which can be applied to any variable or any type to find out how big that type is in the local implementation. This is an important tool professional programmers use to make their programs portable. Figure 4.11 shows how to use `sizeof`. The output from this program will be different on different machines. It depends partly on the hardware and partly on the compiler itself. You should know what the answers are for any system you use. One of the exercises asks you to use `sizeof` to learn about your own compiler's types. The results from the program in Figure 4.11 are shown following the code. From this we can deduce that the workstation is using four bytes for an `int` and eight bytes for a `double`, which is usual. The literal constant `e` is represented as a `double`.

## 4.5 Relational Operators

The **relational operators** (`==`, `!=`, `<`, `>`, `<=`, and `>=`), with their meanings and precedence values, are listed in Figure 4.12. These operators perform comparisons on their operands and return an answer of `true` or `false`. To control program flow, we compare the values of certain variables to each other or to target values. We then use the result to select one of the clauses of an `if` statement or control a loop.

### 4.5.1 True and False

C supports two sets of operators that give true/false results: comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) and logical operators (`&&`, `||`, `!`). The results of these operators are signed integers: `true` is represented by 1 and `false` is represented by 0.

---

All operators listed here associate left to right.

Arity	Usage	Meaning	Precedence
Binary	<code>x &lt; y</code>	Is <code>x</code> less than <code>y</code> ?	10
	<code>x &lt;= y</code>	Is <code>x</code> less than or equal to <code>y</code> ?	10
	<code>x &gt; y</code>	Is <code>x</code> greater than <code>y</code> ?	10
	<code>x &gt;= y</code>	Is <code>x</code> greater than or equal to <code>y</code> ?	10
	<code>x != y</code>	Is <code>x</code> unequal to <code>y</code> ?	9
	<code>x == y</code>	Is <code>x</code> equal to <code>y</code> ?	9

---

**Figure 4.12. Relational operators.**

Every integer has a *truth value*. Values 0 and 1 are the canonical representations of *false* and *true*, but C will interpret any value as true or false if it is used in a context that requires a truth value. The zero values 0 and 0.0 and the zero character, '\0' are all represented by the same bit pattern as 0 and so all are interpreted as false. Any nonzero bit pattern is interpreted as *true*. For example, all of the following are *true*: -1, 2.5, 'F" and 367. Suppose we use one of these values as the entire condition of an `if` statement:

```
if ( -1 ) puts( "true" ) else puts( false" );
```

In human language, this makes no sense. However, to C, it means to find the truth value of -1, which is *true*. So, the answer "true" will be printed. Similarly, if we write

```
if ( 'F' ) puts( "true" ) else puts( false" );
```

The answer "true" will be printed, because 'F' is not a synonym for zero.

Many languages (including Java and C++) have a Boolean type built into the language. It is used to represent the answers to yes-no questions such as, "Is the data value legal?" and "Are two objects the same?" For consistency with C++, the C99 standard introduced a standard type `bool` and defined the boolean values `true` and `false`. To use this type, some compilers require the program to `#include <stdbool.h>`.

A program in which the numeral 1 is used to represent both the number one and the truth value `true` can be difficult to read and interpret. For this reason, it is much better style to write `true` when you mean a truth value and reserve 1 for use as a number. We will use the symbols `false` and `true` for truth values instead of the numbers 0 and 1. We also adopt modern usage and declare truth-valued variables to have type `bool`.

### 4.5.2 The semantics of comparison.

A relational operator can be used to compare two values of any simple type<sup>6</sup> or two values that can be converted to the same simple type. The result is always an `int`, no matter what types the operands are, and it is always either a 1 (`true`) or a 0 (`false`).

A common error among inexperienced C programmers is to use the assignment operator = instead of the comparison operator ==. The expression `x == y` is not the same as `x = y`. The second means "make `x` equal to the current value of `y`", while `x == y` means "compare the values of `x` and `y`". Therefore, if a comparison is done after the inadvertent assignment operation, of course, the values of the two variables *will be* equal. It may help you to pronounce these operators differently; we use "compares equal" for == and "gets" for =.

In other languages, such an error would be trapped by the compiler. In C, however, assignment is an "ordinary" operator that has precedence and associativity like other operators and actually returns a value (the same as the value it stores into memory). A C compiler has no way of knowing for sure whether a programmer meant to write a comparison or an assignment. Some compilers give a warning error comment when the = operator is used within the conditional part of an `if` or `while` statement; however, doing so is not necessarily an error, so the comment is only a warning, not fatal, and the compiler should produce executable

---

<sup>6</sup>Types `double` and `int` are simple. Nonsimple types are compounds with more than one part such as strings, arrays, and structures. These will be discussed in later chapters.

C uses the value 1 to represent `true` and 0 to represent `false`. The result of every comparison and logical operator is either 0 or 1. However, any value can be an input to a logical operator; all nonzero operands are interpreted as `true`. In the table,  $T$  represents any nonzero value.

Operands		Results		
x	y	!x	x && y	x    y
0	0	1	0	0
0	T	1	0	1
T	0	0	0	1
T	T	0	1	1

Figure 4.13. Truth table for the logical operators.

code.

## 4.6 Logical Operators

Sometimes we want to test a more complicated condition than a simple equality or inequality. For example, we might need an input value between 0 and 10 or require two positive inputs that are not equal to each other. We can create compound conditions like these by using the logical operators `&&` (AND), `||` (OR), and `!`(NOT). The former condition can be written as `x >= 0 && x <= 10` and the latter as `x > 0 && y > 0 && x != y`. **Logical operators** let us test and combine the results of comparison expressions.

### 4.6.1 Truth-valued operators.

All the comparison and logical operators produce truth values as their results. For example, if you ask `x == y`, the answer is either 0 (`false`) or 1 (`true`). The meanings of `&&`, `||`, and `!` are summarized by the **truth table** in Figure 4.13. The first two columns of the *truth table* show all possible combinations of the truth values of two operands.  $T$  is used in these columns to mean `true`, because an operand can have any value, not just 1 or 0. The last three columns show the results of the three logical operations. In these columns, true answers are represented by 1 because C always uses the standard `true` to answer questions.

Let us look at a few examples to learn how to use a truth table. Assume `x = 3` and `y = -1`. Then both `x` and `y` are `true`, so we would use the last line of the table. To find the answer for `x && y`, use the fourth column. To find `x || y`, use the fifth column. As a second example, suppose `x=0` and `y=-1`; then `x` is `false` and `y` is `true`, so we use the second row. Therefore, `x || y` is `true` (1) and `x && y` is `false` (0).

### 4.6.2 Parse Trees for Logical Operators

The precedence and usage of the three logical operators are summarized in Figure 4.14. Note that `&&` has higher precedence than `||` and that both are quite low in the precedence table. If an expression combines arithmetic, comparisons, and logic, the arithmetic will be parsed first, the comparisons second, and the logic last. The practical effect of this precedence order is that we can omit many of the possible parentheses in expressions. Figure 4.15 gives an example of how to parse an expression with operators of all three kinds. Before beginning the parse, we note the precedence of each operator used. Beginning with the highest-precedence operator and proceeding downward, we then parenthesize or bracket each operator with its operands.

A small circle, called a **sequence point**, is written on the parse tree under every `&&` and `||` operator. The order of evaluation of the parts of a logical expression is different from other expressions, and the sequence points remind us of this difference. For these operators, the part of the tree to the left of the sequence point is always evaluated before the part on the right.<sup>7</sup> This fact is very important in practice because it permits us to use the left side of a logical expression to “guard” the right side, as explained in the next section.

<sup>7</sup> Two other operators, the question mark and the comma, have sequence points associated with them. For all other operators in C, either the right side of the tree or its left side may be evaluated first.

Arity	Usage	Meaning	Precedence	Associativity
Unary	<code>!x</code>	logical-NOT x (logical opposite)	15	right to left
Binary	<code>x &amp;&amp; y</code>	x logical-AND y	5	left to right
	<code>x    y</code>	x logical-OR y	4	"

Figure 4.14. Precedence of the logical operators.

### 4.6.3 Lazy Evaluation

Logical operators have a special property that makes them different from all other operators: You often can know the result of an operation without even looking at the second operand. Look again at the truth table in Figure 4.13, and note that the answer for `x && y` is always 0 when `x` is `false`. Similarly, the answer for `x || y` is always 1 when `x` is `true`. This leads to a special method of computation, called **lazy evaluation**, that can be used only for logical operators. Basically, the left operand of the logical operator is evaluated then tested. If that alone decides the value of the entire expression, the rest is skipped. We show this skipping on the parse tree by writing a diagonal *pruning mark* on the branch of the tree that is skipped. You can see these marks on the trees in Figures 4.17 and 4.19. To further emphasize the skipping, a loop is drawn around the skipped portion. Note that *no operator* within the loop is executed.

---

We parse the expression `y = a < 10 || a >= 2 * b && b != 1` using parentheses and a tree.

Parenthesizing in precedence order:

- Level 13: `y = a < 10 || a >= (2*b) && b != 1`
- Level 10: `y = (a < 10) || (a >= (2*b)) && b != 1`
- Level 9: `y = (a < 10) || (a >= (2*b)) && (b != 1)`
- Level 5: `y = (a < 10) || ((a >= (2*b)) && (b != 1))`
- Level 4: `y = (((a < 10) || ((a >= (2*b)) && (b != 1))))`
- Level 2: `(y = (((a < 10) || ((a >= (2*b)) && (b != 1)))))`

Precedence level is listed above each operator.

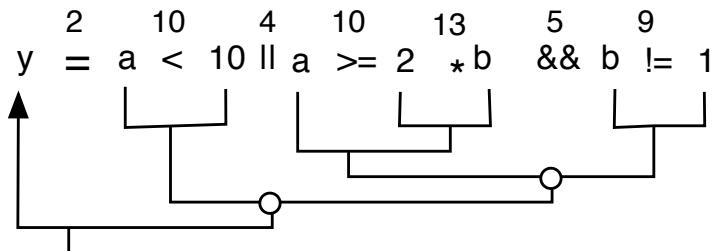


Figure 4.15. Parsing logical operators.

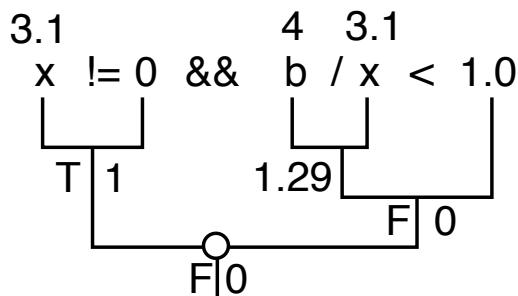
x	y	x && y
0	?	0 and skip second operand
T	0	0
T	T	1

- To evaluate an `&&` operator, first evaluate its *left* operand. This operand might be a simple variable or literal, or it might be a complicated expression.
- Look at the truth value. If it is `false`, return 0 as the answer to the `&&` operation and skip the next step.
- Otherwise, we do not yet know the outcome of the expression, so evaluate the right operand. If it is `false`, return 0. Otherwise, return 1.

Figure 4.16. Lazy truth table for logical-AND.

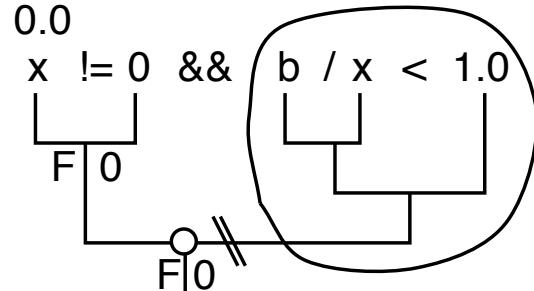
We evaluate a logical expression twice with different operand values.

(1) Evaluation with  $x = 3.1$  and  $b = 4$ . All parts of the expression are evaluated because the left operand is `true`.



The `/` and `<` operators are evaluated and their results are written under the operators on the tree.

(2) Evaluation with the values  $x = 0.0$  and  $b = \text{anything}$ . Skipping happened because the left operand is `false`.



The “pruning mark” on the tree and the looped line show the part of the expression that was skipped.

**Figure 4.17. Lazy evaluation of logical-AND.**

**Logical-AND.** Figure 4.16 summarizes how lazy evaluation works for logical-AND, and Figure 4.17 illustrates the most important use of lazy evaluation: guarding an expression. The left side of a logical-AND expression can be used to “guard” the right side. We use the left side to check for “safe” data conditions; if found, the left side is `true` and we execute the right side. If the left side is `false`, we have identified a data value that would cause trouble and use lazy evaluation to avoid executing the right side. In this way, we avoid computations that would cause a machine error or program malfunction. For example, in Figure 4.17, we want to test a condition with  $x$  in the denominator. But dividing by 0 is an error, so we should check the value of  $x$  before testing this condition. If it is 0, we skip the rest of the test; if it is nonzero, we go ahead.

**Logical-OR.** Figure 4.18 summarizes how lazy evaluation works for logical-OR, and Figure 4.19 illustrates how lazy evaluation can be used to improve program efficiency. Logical-OR often is used for data validation. If one validity condition is fast to compute and another is slow, we can save a little computation time by putting the fast test on the left side of the logical-OR and the slow test on the right. Or we could put the most common

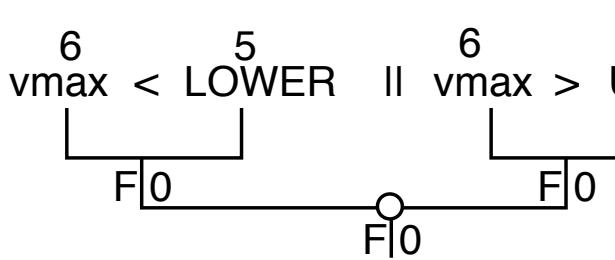
x	y	$x \text{    } y$
T	?	1 and skip second operand
0	0	0
0	T	1

- To evaluate an `||` operator, first evaluate its *left* operand. This operand might be a simple variable or literal or it might be a more complicated expression.
- Look at the truth value. If it is `true`, return 1 as the answer to the `||` operation and skip the next step.
- Otherwise, we do not yet know the outcome of the expression so evaluate the right operand. If it is `false`, return 0. Otherwise, return 1.

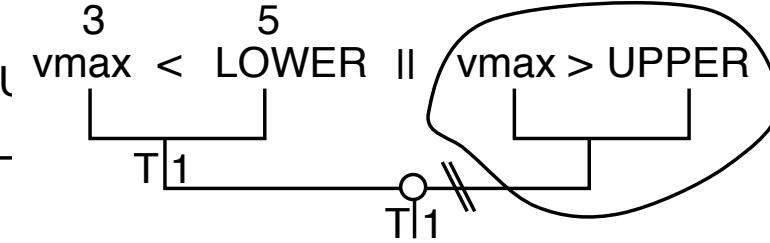
**Figure 4.18. Lazy truth table for logical-OR.**

We evaluate a logical expression twice with different operand values. The first time, the entire expression is evaluated. The second time, a shortcut is taken.

(1) Evaluation with  $vmax = 6$ . All parts of the expression are evaluated because the left operand is **false**.



(2) Evaluation with  $vmax = 3$ . The left side is **true**, which causes skipping.



The  $<$  and  $>$  operators are both evaluated and their results are written under the operators on the tree.

The pruning mark on the tree and the looped line show the part of the expression that was skipped.

Figure 4.19. Lazy evaluation of logical-OR.

problem on the left and an unusual problem on the right.

Figure 4.19 shows two evaluations of a logical-OR expression that is used in the next program. If the input data fails the first test, the second test determines the result of the expression. If the data passes the first test, we save time by skipping the second test and return the result of the left side.

## 4.7 Integer Operations

Modern computers have two separate sets of machine instructions that perform arithmetic: one for integers, the other for floating-point numbers. We say that the operators,  $*$ ,  $/$ ,  $+$ , and  $-$ , are *generic*, because they have more than one possible translation. When a C compiler translates a generic operator, it uses the types of the operands to select either an integer or a floating-point operation. The compiler will choose integer operations when both operands are integers; the result will also be an integer. In all other cases, a floating-point operation will be chosen.

### 4.7.1 Integer Division and Modulus

**Division.** Like the other arithmetic operators, the division operator  $/$  is generic; its meaning depends on the types of its operands. However, each of the operators  $*$ ,  $+$ , and  $-$  symbolizes a single mathematical operation, even though it is performed in two different ways on the two kinds of number representations. In contrast,  $/$  represents two different mathematical operations: **real division** (where the answer has a fractional part) and **integer division** (where there are two parts of the answer, the quotient and the remainder). Here, the instruction used makes a significant difference. With floating-point operands, only real division is meaningful. However, with integer operands, both real and integer division are useful and a programmer might wish to use either one. In C, if both operands are integers, the integer quotient is calculated and the remainder is forgotten. A programmer who needs the remainder of the answer can use the modulus operator,  $\%$ , which is described shortly. If the entire fractional answer is needed, one of the operands must first be converted to floating-point representation. (Data type conversion techniques are covered later in this chapter.)

$x$	$y$	$x \% y$	$x$	$y$	$x \% y$
10	3	1	3	10	3
9	3	0	3	9	3
8	3	2	7	2873	7
7	3	1	7	7	0
6	3	0	7	1	0
5	3	2	7	0	Undefined

Figure 4.20. The modulus operation is cyclic.

**Division by 0.** Attempting to divide by 0 will cause an error that the computer hardware can detect. In most cases, this error will cause the program to terminate.<sup>8</sup> It always is wise to check for this condition before doing the division operation in order to make the program as robust as possible; that is, it does something sensible even when given incorrect data.

**Indeterminate answers (optional topic).** A further complication of integer division is that the C standard allows two different correct answers for  $x/y$  in some cases. When  $x$  is not an even multiple of  $y$  and either is negative, the answer can be the integer either just larger than or just smaller than the true quotient (this choice is made by the compiler, not the programmer). This indeterminacy is provided by the standard to accommodate different kinds of hardware division instructions. The implication is that programs using division with signed integers may be nonportable because the answer produced depends on the hardware. This “feature” of the language is worse in theory than in practice; we tested C compilers running on a variety of hardware platforms and found that they all truncate the answer toward 0 (rather than negative infinity). However, the careful C programmer should be aware of the potential problem here.

**Modulus.** When integer division is performed, the answer has two parts: a quotient and a remainder. C has no provision for returning a two-part answer from one operation, so it provides two operators. The **integer modulus** operator, named *mod* and written `%`, performs the division and returns the remainder, while `/` returns the quotient. Figure 4.20 shows the results of using `%` for several positive values; Figure 4.21 is a visual presentation of how mod is computed. Note the following properties of this operator:

- Mod is defined for integers  $x$  and  $y$  in terms of integer division as:  $x \% y = x - y \times (x/y)$ .
- If  $x$  is a multiple of  $y$ , the answer is 0.
- If  $x$  is smaller than  $y$ , the answer is  $x$ .
- The operation  $x \% y$  is a cyclic function whose answer (for positive operands) always is between 0 and  $y - 1$ .
- This operator has no meaning for floating-point operands.
- If  $y$  is 0,  $x \% y$  is undefined. At run time a division by 0 error will occur.
- The results of `/` with negative values is not fully defined by the standard; implementations may vary. For example,  $-5/3$  can equal either  $-1$  (the usual answer) or  $-2$ . Since the definition of `%` depends on the definition of `/`, the result of  $x \% y$  is indeterminate if either  $x$  or  $y$  is negative. Therefore,  $-5 \% 3$  can equal either  $-2$  or  $1$ .

A program that uses integer `/` and `%` is given in the next section, and one that uses the `%` operator to help format the output into columns is in Figure 5.26.

### 4.7.2 Applying Integer Division and Modulus

We normally count and express numbers in base 10, probably because we have 10 fingers. However, any number greater than 1 can be used as the base for a positional notation.<sup>9</sup> Computers use base 2 (binary) internally

<sup>8</sup>More advanced techniques, beyond the scope of this book, can be used to take special action after the termination.

<sup>9</sup>Theoretically, negative bases can be used, too, but they are beyond the scope of this text.

---

To calculate  $a \% b$ , we distribute  $a$  markers in  $b$  columns. The answer is the number of markers in the last, partially filled row. (If the last row is filled, the answer is 0.)

operation:	$12 \% 5$	$15 \% 5$	$4 \% 5$	$5 \% 4$	$10 \% 4$
	x x x x x	x x x x x	x x x x .	x x x x	x x x x
	x x x x x	x x x x x		x . . .	x x x x
	x x . . .	x x x x x			x x . .
answer:	1 2 3 4 0	1 2 3 4 0	1 2 3 4 0	1 2 3 0	1 2 3 0
	↑	↑	↑	↑	↑

---

Figure 4.21. Visualizing the modulus operator.

and the C language lets us write numbers in bases 8 (octal) and 16 (hexadecimal) as well as base 10 (decimal). These number representations and the simple algorithms for converting numbers from one base to another are described in Appendix E. The next program shows how one can use a computer to convert a number from its internal representation (binary) to any desired base. The algorithm used is based on the meaning of a positional notation:

- Each digit in a number represents a multiple of its place value.
- The place value of the rightmost position is 1.
- The place value of each other position is the base times the place value of the digit to its right.

Therefore, given a number  $N$  (in the computer's internal representation) and a base  $B$ ,  $N \% B$  is the digit whose place value is  $B^0 = 1$  when expressed in positional notation using base  $B$ . We can use these facts to convert a number  $N$  to the equivalent value  $N'$  in base  $B$ .

The algorithm is a simple loop that generates the digits of  $N'$  from right to left. On the first pass through the conversion loop, we compute  $N \% B$ , which is the rightmost digit of  $N'$ . Having done so, we are no longer interested in the 1's place, so we compute  $N = N/B$  to eliminate it and prepare for the next iteration. We continue this pattern of using  $\%$  to generate digits and integer division to reduce the size of  $N$  until nothing is left to convert. An example is given in Figure 4.22, a program that implements this algorithm is in Figure 4.22 and the flow diagram for this program in Figure 4.24. This program also illustrates the use of an increment operator, logical opertor, and assignment combination operator.

#### Notes on Figure 4.23. Number conversion.

---

Any number  $N$  can be expressed in positional notation as a series of digits:

$$N = \dots D_3 D_2 D_1 D_0$$

If the number's base is  $B$ , then each digit is between 0 and  $B - 1$  and the value of the number is

$$N = \dots B^3 \times D_3 + B^2 \times D_2 + B^1 \times D_1 + D_0$$

Now, if  $N = 1234$  and  $B = 10$ , we can generate all the digits of  $N$  by repeatedly taking  $N \% B$  and reducing  $N$  by a factor of  $B$  each time:

$$\begin{array}{ll} D_0 = 1234 \% 10 = 4 & N_1 = 1234/10 = 123 \\ D_1 = 123 \% 10 = 3 & N_2 = 123/10 = 12 \\ D_2 = 12 \% 10 = 2 & N_3 = 12/10 = 1 \\ D_3 = 1 \% 10 = 1 & N_4 = 1/10 = 0 \end{array}$$


---

Figure 4.22. Positional notation and base conversion.

***First box: selecting a valid base and the number to convert.***

- We have restricted the acceptable number bases to the range 2...10, which restricts the possible digits in the answer to the range 0...9. This algorithm could be used to convert a number to any base. We demonstrate it here only for bases of less than 10 because we wish to focus attention on the conversion algorithm, not on the representation of digits greater than 9.
- We use 2 as the minimum base value; 1 and 0 cannot be used as bases for a positional notation. The following output shows an example of error handling:

```
Read an integer and express in a given base.
Please enter a number to convert and
a target base between 2 and 10: 3 45
Base must be between 2 and 10.
```

- Any integer, positive, negative or zero, can be accepted as input for the number to be converted. However, 0 must be treated as a special case. The following output shows an example

```
Read an integer and express in a given base.
Please enter a number to convert and
a target base between 2 and 10: 0 3
0 is 0 in every base.
```

Convert an integer to a selected base and print it using place-value notation.

```
#include <stdio.h>
int main( void )
{
    int n;           // input: the number to convert
    int base;        // input: base to which we will convert n
    int rhdigit;    // right-hand digit of n-prime
    int power;       // loop counter

    printf( " Read an integer and express it in a given base.\n"
            " Please enter a number to convert and\n"
            " a target base between 2 and 10: " );
    scanf( "%i %i", &n, &base );

    if (base < 2 || base > 10) printf( " Base must be between 2 and 10\n" );
    else if (n==0) printf ( " 0 is 0 in every base.\n" );

    power = 0;
    // --- Generate and print digits of converted number, right to left.
    while (n != 0) {
        if (power == 0) printf( "%12li = ", n );
        else printf( "         +   ");

        rhdigit = n % base;      // Isolate right-hand digit of n.
        n /= base;               // then eliminate right-hand digit.

        printf( "%hi * %hi^%i \n", rhdigit, base, power );
        ++power;
    }

    return 0;
}
```

Figure 4.23. Number conversion.

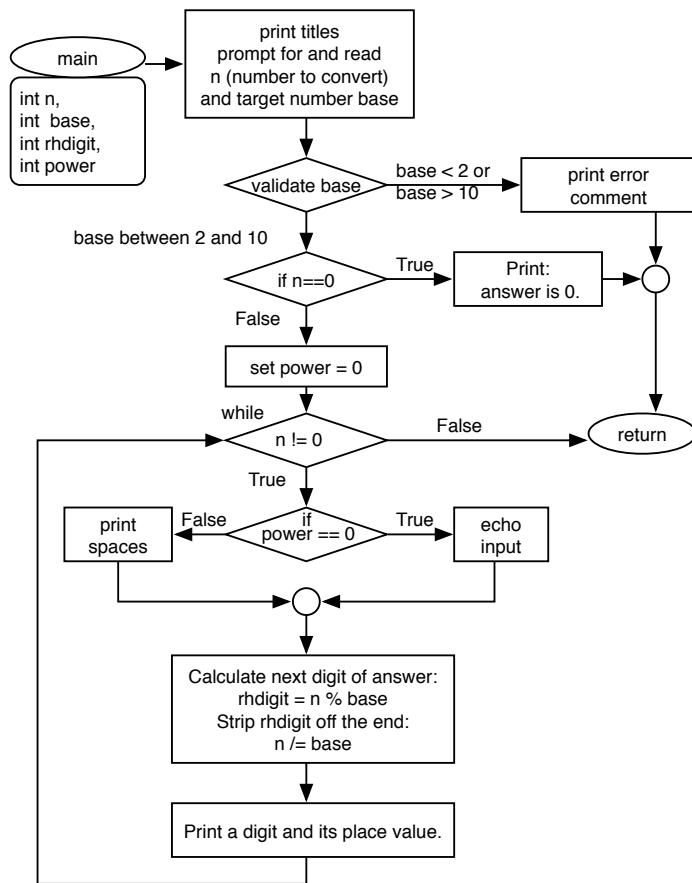


Figure 4.24. A flow diagram for the base conversion.

**Second box (outer): converting to the selected base.**

- We start by generating the coefficient of  $B^0$ , so we set `power = 0`. After each iteration, we increment `power` to prepare for converting the coefficients of  $B^1$ ,  $B^2$ , and so forth.
- We reduce the size of  $N$  on each iteration by dividing it by the target base; we continue until  $N$  is reduced to 0.
- Note that a loop test need not always involve the loop counter; it is necessary only that the value being tested change somewhere within the loop.
- We first print the number itself (in base 10) and then an = sign before the first term of the answer. We indent and print a + before all other terms.

**Two examples of the output.**

```

Read an integer and express in a given base.
Please enter a number to convert and
a target base between 2 and 10: 45 3

```

```

45 =   0 * 3^0
      + 0 * 3^1
      + 2 * 3^2
      + 1 * 3^3

```

```

Read an integer and express in a given base.
Please enter a number to convert and
a target base between 2 and 10: -45 10

```

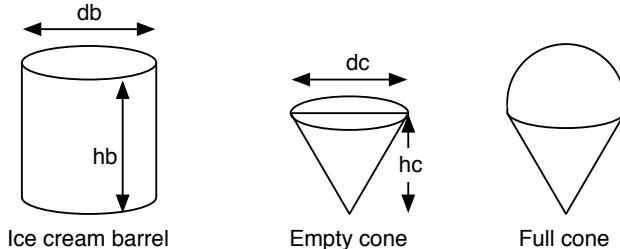
```

-45 =  -5 * 10^0
      + -4 * 10^1

```

**Problem scope:** Determine how many barrels of ice cream to buy to fill one cone for each guest at the annual picnic. The cone portion should be filled, and a half-sphere of ice cream should be on top.

**Inputs:** Diameter and height of cones. Diameter and height of a 3-gallon barrel of ice cream. (All in cm.)



**Constants:** Number of guests (number of cones needed).

**Formulas:**

$$\begin{aligned}
 \text{volume of ice cream needed} &= \text{number of guests} * \text{volume of one full cone} \\
 \text{barrels needed} &= \text{volume of ice cream needed} / \text{volume of barrel} \\
 \text{volume of full cone} &= \text{volume of hemisphere} + \text{volume of empty cone} \\
 \text{volume of barrel} &= \pi * hb * db^2 / 4 \\
 \text{volume of cone} &= \pi * (dc/2)^2 * hc / 3 \\
 \text{volume of hemisphere} &= 2 * \pi * (dc/2)^3 / 3
 \end{aligned}$$

**Output required:** Number of 3-gallon barrels of ice cream to buy.

**Debugging outputs:** Volume of one barrel, volume of an empty cone, volume of the half-sphere of ice cream on top of the cone, and total volume of the full cone.

Figure 4.25. Problem specification: Ice cream for the picnic.

**Inner box: decomposing the number, digit by digit.** As explained, we use % to generate each digit of  $N'$ . Then we use integer division to reduce  $N$  by removing the extracted digit and shifting the others one position to the right. This prepares  $N$  for the next iteration. Real division would not work here; the algorithm relies on the fact that the remainder is discarded.

## 4.8 Techniques for Debugging

### 4.8.1 Using Assignments and Printouts to Debug

One cause for logic errors is complexity, and that complexity also makes the errors difficult to find and difficult to correct. Sometimes an algorithm is complex; that can be addressed by breaking it into modular parts, as shown in Chapter 5. Here, we address the problem of debugging complex formulas.

Suppose we wish to write a program for the specification in Figure 4.25. We could combine all the formulas given into one line:

$$\text{barrels needed} = \text{Number of guests} * \pi * (dc^3 / 12 + (dc/2)^2 * hc / 3) / (hb * db^2 / 4)$$

However, the resulting formula is quite complex and difficult to compute by hand or in one's head. It would make much more sense to calculate each formula, as given. Even better, you might notice that the expression  $dc/2$  is used twice, and compute it separately also. The program in Figure 4.26 uses a sequence of assignment statements to calculate the parts of the long, messy formula.

Notes on Figure 4.26. How much ice cream?

***First box.***

In addition to the four variables that we need for input and the one for output, we define four variables for intermediate results.

***Second box.***

An important aid in debugging is to be certain that the inputs that were actually read by the program are the ones that the user intended to enter. Echoing the inputs as part of the output is sound programming practice.

***Third box.***

- We implement the separate formulas instead of the combined formula to make both programming and debugging easier and less error prone. It is a little more work for the computer this way, but in almost all circumstances, simplicity is better than complexity.
- We compute each formula and store its result for later use. We also print each result, so that the user can hand-verify each part of the computation.
- These formulas are not computed in the same order as they are given in the specification because, in C,

```
#include <stdio.h>
#define GUESTS 100
#define PI 3.1415927

int main( void )
{
    double h_cone, d_cone;      // Height and diameter of part
    double h_barrel, d_barrel;  // Height and diameter of barrel
    double n_barrels;          // Number of barrels needed.

    double r_cone, v_cone;      // Cone's radius and volume
    double v_barrel, v_hemi;    // Volume of barrel and ice cream on top.

    printf( " How much ice cream do we need?\n"
            " Enter Diameter and height of ice cream barrel: " );
    scanf( "%lg %lg", &d_barrel, &h_barrel );
    printf( " Enter Diameter and height of an empty cone: " );
    scanf( "%lg %lg", &d_cone, &h_cone );

    printf( "      Barrel is %g cm. wide, %g cm. tall.\n", d_barrel, h_barrel );
    printf( "      Cones are %g cm. wide, %g cm. tall.\n", d_cone, h_cone );

    v_barrel = PI * h_barrel * d_barrel * d_barrel / 4;
    printf( "      Barrel volume = %g\n", v_barrel );
    r_cone = d_cone / 2;
    v_cone = PI * r_cone * r_cone * h_cone / 3;
    printf( "      Cone volume = %g\n", v_cone );
    v_hemi = 2 * PI * r_cone * r_cone * r_cone / 3;
    printf( "      Top volume = %g\n", v_hemi );
    printf( "      Full cone volume = %g cm^3\n", v_hemi + v_cone );

    n_barrels = GUESTS * (v_hemi + v_cone) / v_barrel ;
    printf( " You need %g barrels of ice cream.\n", n_barrels );

    return 0;
}
```

Figure 4.26. How much ice cream?

each part must be computed before it is used. The specification starts with the general formula and goes on to the details. We must compute the details before we can compute the general formula.

- We want to print the volume of the full cone, but do not need it in any further computation. Moreover, the formula is simple (it involves only one operation). So we choose to write the formula as part of the `printf()` statement, rather than as a separate assignment statement. This technique is good style as long as the results are not very complex.
- Sample output:

```
How many gallons of ice cream do we need?
Enter Diameter and height of ice cream barrel: 30 40
Enter Diameter and height of an empty cone: 10 15
    Barrel is 30 cm. wide, 40 cm. tall.
    Cones are 10 cm. wide, 15 cm. tall.
    Barrel volume = 28274.3 cm^3
    Cone volume = 392.699 cm^3
    Top volume = 261.799 cm^3
    Full cone volume = 654.498 cm^3
    You need 2.31481 barrels of ice cream.
```

- If you have an on-line debugger and know how to use it, you can set a breakpoint after each computation instead of printing the result. Both ways will give you the information you need. However, if you need to ask someone who is not present for assistance, it is essential to provide that person with printouts of both the program and the results.

#### *Fourth box.*

- Finally we are able to implement one form of the general formula. It is too long and complex to fit easily into a `printf()` statement, so we compute the answer and store it in a variable first, then print it.
- When we wrote this program, we were uncertain whether the answers were right or wrong, So we used the intermediate printouts with a calculator to verify that each part was correct and made sense. Then we verified that the final result was correctly computed from the intermediate results.

### 4.8.2 Case Study: Using a Parse Tree to Debug

When a program seems to work but gives the wrong answers, the problem sometimes lies in the expressions that calculate those answers. Drawing a parse tree can help **debug** the program; that is, help a programmer find the error. We illustrate this technique through a case study for which Figure 4.27 gives the full specification. In this problem, a circuit is wired with three resistances connected in parallel, as shown in the specification. We must calculate the electrical resistance equivalent,  $r_{eq}$ , for this part of the circuit.

**Step 1. Making a test plan.** Making a **test plan** first is a good way to understand the problem. It forces us to analyze the formulas, look at the details, and think about what kind of data might cause problems. We look at the problem specification to decide what the test plan should be.

The first test case should be something that can be computed in one's head. We note that the arithmetic is very simple if all the resistances are 2 ohms, so we enter this case in the test chart, which follows. We want to test inputs with fractional values and note that we can easily compute the answer for three resistances of 0.1 ohm each. Then we notice that, if two inputs are 0, the denominator of the fraction will be 0. Since division by 0 is undefined, this will cause trouble. Since no limitations on input values are specified, it is unclear what to do about this. Resistances that are 0 or negative make no realistic sense, so we decide to warn the user and trust him or her to enter valid data. Next, we enter an arbitrary set of values, just to see what the output will look like for the typical case. We use a pocket calculator and a pencil to do the computation by hand. We now have three tests in our test plan, which is enough for a very simple program.

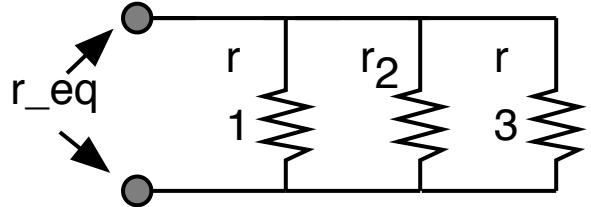
$r_1$	$r_2$	$r_3$	$r_{eq}$
2	2	2	$8.0/12.0 = 0.666667$
0.1	0.1	0.1	$0.001/0.03 = 0.033333$
75	40	2.5	2.28137

**Problem scope:** Find the electrical resistance equivalent,  $r_{eq}$ , for three resistances wired in parallel.

**Input:** Three resistance values,  $r_1$ ,  $r_2$ , and  $r_3$ .

**Formula:**

$$r_{eq} = \frac{r_1 * r_2 * r_3}{r_1 * r_2 + r_1 * r_3 + r_2 * r_3}$$



**Constants:** None.

**Output required:** The three inputs and their equivalent resistance.

**Computational requirements:** The equivalent resistance should be a real number, not an integer.

Figure 4.27. Problem specification: Computing resistance.

**Step 2. Starting the program.** Write the parts that remain the same from application to application. We write the `#include` statement and the first and last lines of `main()` with the opening and closing messages. The dots (...) represent the unfinished parts of the program.

```
#include <stdio.h>
...
int main( void )
{
    ...
    puts( "\n Computing Equivalent Resistance \n" );
    ...
    puts( "\n Normal termination." );
}
```

**Step 3. Reading the input.** We need to read three resistance values; we could do this with three calls on `scanf()` or with one. We choose to use one call because one input step is faster for the user and we do not think the user will be confused by giving three answers for one prompt in this situation. To store the three values, we need three variables, which we name  $r_1$ ,  $r_2$ , and  $r_3$ . We declare these as type `double` (not `int`) because the answer will have a fractional part. We write a declaration for three doubles:

```
double r1, r2, r3; // input variables for resistances
```

Now we are ready for the prompt and the input. In the format for `scanf()`, we write three percent signs because we will be reading three values. Since we are reading `double` values, we write `lg` (the letter l, not the numeral 1) after the percent signs. We remember to write the ampersand before the name of each variable that needs to receive an input value.

```
printf( "\n Enter resistances #1, #2, and #3 (ohms).\n"
        " All resistances must be greater than 0: " );
scanf( "%lg%lg%lg", &r1, &r2, &r3 );
```

Finally, we write a `printf()` statement to echo the three input values. In the output format, we again write three percent signs, for three values. However, the correct output code for type `double` is `g`, without the `l`. At the end of the format we remember to write a newline character. After the format we list the names of the variables to print, without ampersands. (Reading into a variable requires an ampersand; writing does not.)

```
printf( "\n     r1= %g     r2= %g     r3= %g\n", r1, r2, r3 );
```

---

```

// -----
// Compute the equivalent resistance of three resistors in parallel
//
#include <stdio.h>
int main( void )
{
    double r1, r2, r3; // input variables for three resistances
    double r_eq; // equivalent resistance
    puts( "\n Computing Equivalent Resistance\n" );
    printf( "\n Enter values of resistances #1, #2, and #3 (ohms).\n"
            " All resistances must be greater than 0: " );
    scanf( "%lg%lg%lg", &r1, &r2, &r3 );
    printf( "    r1= %g    r2= %g    r3= %g\n", r1, r2, r3 );

    r_eq = r1 * r2 * r3 / r1 * r2 + r1 * r3 + r2 * r3;
    printf( " The equivalent resistance is %g\n\n", r_eq );

    return 0;
}

```

---

Figure 4.28. Computing resistance.

**Step 4. Computation and output.** Now we transcribe the mathematical formula into C notation, changing the fraction bar to a division sign and writing the subscripts as part of the variable names. In the process, we note that we need a variable for the result and declare another `double`. Then we write a `printf()` statement to print the result. We have

```

double r_eq; // equivalent resistance
...
r_eq = r1 * r2 * r3 / r1 * r2 + r1 * r3 + r2 * r3;
printf( " The equivalent resistance is %g\n", r_eq );

```

**Step 5. Putting it together and testing it.** We now type in all the parts of the program. The code compiled successfully after correcting a few typographical errors; the result is shown in Figure 4.28. We then ran the program and entered the first data set. The output was

```

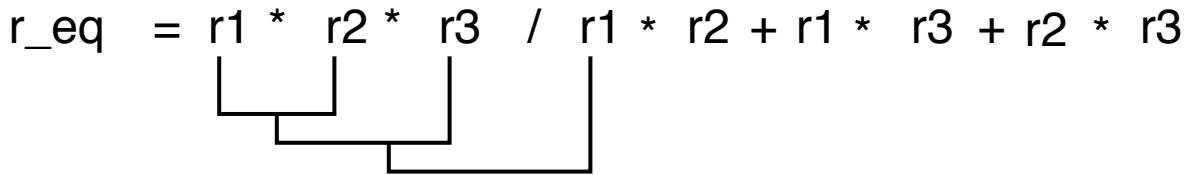
Computing Equivalent Resistance
Enter values of resistances #1, #2, and #3 (ohms).
All resistances must be greater than 0: 2 2 2
    r1= 2    r2= 2    r3= 2
The equivalent resistance is 16

```

Comparing the answer to the answer in our test plan, we see that it is wrong. The correct answer is 0.667. What could account for the error? There are three possibilities:

1. The input was read incorrectly. This could happen if the format were inappropriate for the data type of the variable or if we forgot to write an ampersand in front of the variable name.
2. The answer was printed incorrectly. This could happen if the format were inappropriate for the data type of the variable or if we wrote the wrong variable name or put an ampersand in front of it.
3. The answer was computed incorrectly.

We eliminate the first possibility immediately; we echoed the data and know it was read correctly. This is why every program should echo its inputs. Then we look carefully at the final `printf()` statement and see no




---

Figure 4.29. Finding an expression error.

errors. We think this is not the problem. The remaining possibility is that the computation is wrong, so we need to analyze the formula we wrote.

**Step 6. Correcting the error.** The best way to analyze a computation is with a parse tree, so we copy the expression on a piece of paper and begin drawing the tree (see Figure 4.29).

1. The  $*$  and  $/$  are the highest precedence operators in the expression so they are parsed first using left-to-right associativity, as shown in Figure 4.29.
2. At the  $/$  sign, we see that the right operand is  $r1$ , which does not correspond to the mathematical formula. The error becomes clear; the denominator for  $/$  should be the entire subexpression  $r1 * r2 + r1 * r3 + r2 * r3$ , not just  $r1$ .
3. The error is corrected by adding parentheses as shown in Figure 4.30, so that it now corresponds to the mathematical formula.

We correct the program, recompile it, and retest it. The results are

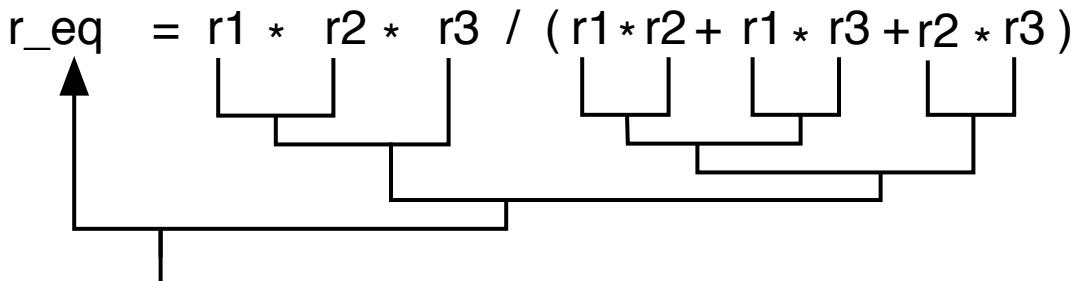
```
Computing Equivalent Resistance
Enter values of resistances #1, #2, and #3 (ohms).
All resistances must be greater than 0: 2 2 2
r1= 2    r2= 2    r3= 2
The equivalent resistance is 0.667

Normal termination.
```

We see that the first answer is now correct. Before going on, we also consider the appearance of the output. Is it neat and readable? Does every number have a label? Should the vertical or horizontal spacing be adjusted? We decide to add a blank line before the final answer, so we insert a `\n` at the beginning of the format. The boxed section of the program now is

```
r_eq = r1 * r2 * r3 / (r1 * r2 + r1 * r3 + r2 * r3);
printf( "\n The equivalent resistance is %g\n", r_eq );
```

---




---

Figure 4.30. Parsing the corrected expression.

We recompile the program and go on with the test plan. The program produces correct results for the other two test cases:

```
Enter values of resistances #1, #2, and #3 (ohms).
All resistances must be greater than 0: .1 .1 .1
r1= 0.1    r2= 0.1    r3= 0.1

The equivalent resistance is 0.033
-----
Enter values of resistances #1, #2, and #3 (ohms)
All resistances must be greater than 0: 75 40 2.5
r1= 75     r2= 40     r3= 2.5

The equivalent resistance is 2.281
```

The output is correct, neat, and readable, so we declare the program finished.

## 4.9 What You Should Remember

### 4.9.1 Major Concepts

This chapter is concerned with how to create and name objects, how to use types to describe their properties, and how to combine the objects with operators to form expressions. These concepts are summarized here.

- Operators:
  - Operators are like verbs: They represent actions and can be applied to objects of appropriate types. An expression is like a sentence: It combines operators with the names of objects to specify a computation.
  - Precedence, associativity, and parentheses control the structure of an expression. The precedence of C operators follows normal mathematical conventions.
  - A few operators have side effects; that is, they modify the value of some variable in memory. These include the assignment operator, assignment combinations, increment, and decrement.
  - Integer division is not the same as division using real numbers; any remainder from an integer division is forgotten. The remainder, if needed, must be computed by using the modulus operator (%).
  - The result of a comparison is always either `true` (1) or `false` (0).
  - Every data value can also be interpreted as either `true` or `false`: zero is `false`, and every other value is `true`. Sometimes it surprises beginners to learn that a negative number will be interpreted as `true` when it is the operand of a logical operator, an `if` statement or a `while` statement.
- Diagrams. Diagrams are used to visualize the parts of a program and how they interact. They become increasingly important as the more complex features of C are introduced. We have now introduced three ways to visualize the aspects of a program:
  1. A flow diagram (introduced in Chapter 3) is used to depict the structure of an entire program and clarify the sequence of execution of its statements.
  2. A parse tree is used to show the structure of an expression and can be used to manually evaluate the expression.
  3. An object diagram is used to visualize a variable. Simple object diagrams were introduced in this chapter; more elaborate diagrams will be introduced as new data types are presented.
- Debugging. To debug a program, you must find and correct all the syntactic, semantic, and logical errors. The best practice is to design and write your code so that this becomes easy:
  1. Strive for simplicity at all times in every way.
  2. Do a thorough specification first.
  3. Keep your formulas short.
  4. Echo the inputs.
  5. Print out intermediate results or use an online debugger.
  6. Hand-check the results.
  7. Use a parse tree if you cannot find an error in a formula.

### 4.9.2 Programming Style

- Names: You must name every object and function you create. The compiler does not care what names you use as long as they are consistent. However, people do care. Obscure names, silly names, and unpronounceable names hinder comprehension. A program with bad names takes longer to debug.
- The length of a name: Extremely long and short names are poor choices. Except in unusual circumstances, a one- or two-letter name does not convey enough information to clarify its meaning. At the other extreme, very long, wordy names are distracting and often obscure the structure of an expression.
- Long expressions: Very long, complex expressions are difficult to write correctly and difficult to debug. When a formula is long and complex or has repeated subexpressions, it is a good idea to break it into several separate assignment statements.
- Parentheses: Use parentheses to clarify the structure of your expressions by enclosing meaningful subexpressions. Use them when you are uncertain about the precedence of operators. However, use parentheses sparingly; too many can be worse than too few. When three and four parentheses pile up in one part of an expression, they can be hard to “pair up” visually. In this situation, moderation is the key to good style.
- Increment and decrement operators: These operators can give nonintuitive results because of C’s complicated rules about the order in which parts of an expression are evaluated. Until you fully understand the evaluation rules, restrict your use to very short expressions and avoid combining these operators with logical `&&` and `||`.
- When using division or modulus, be sure that there is no possibility that the divisor is 0. Dividing by 0 causes an immediate program crash in many systems and produces incorrect results on others. If a 0 divisor is possible, test for it.

### 4.9.3 New and Revisited Vocabulary

*The most important terms and concepts discussed in this chapter:*

garbage	parse tree	truth value
precedence	test plan	truth table
associativity	debugging	lazy evaluation
arity	arithmetic operators	increment operators
precedence table	assignment operator	postfix operator
operator	combination operators	prefix operator
binary operator	side effect	integer arithmetic
operand	relational operators	modulus
expression	logical operators	intermediate printouts

*C keywords and operators introduced or discussed in this chapter:*

<code>sizeof</code>	<code>=, +=, -=, *=, /=</code>	<code>++x</code> (preincrement)
<code>(...)</code>	<code>&lt;, &lt;=, &gt;, &gt;=</code>	<code>x++</code> (postincrement )
<code>+, -, *, /</code>	<code>==, !=</code>	<code>--x</code> (predecrement )
<code>integer /, %</code>	<code>&amp;&amp;,   , !</code>	<code>x--</code> (postdecrement )

### 4.9.4 Sticky Points and Common Errors

This has been a long chapter, filled with many facts about C semantics and C operators. The table in Figure 4.31 gives a brief review of the difficult aspects of C operators to assist you in program planning and debugging.

### 4.9.5 Where to Find More Information

- Program 4.A on the Applied C website shows a typical use of preincrement in a counting loop.
- The C operators for types `int` and `double` were described in this chapter. Operations on characters will be explained in Chapter 8 and operations on bits are in Chapter 15.

Group	Operators	Complication
Arithmetic	/	Division by 0 or 0.0 is undefined.
	/	Integer division is used if both operands are integers; the result is an integer. The fractional part is discarded.
	%	Not defined for floating-point values. For integers, the result is the remainder of an integer division.
	/, %	If both arguments are integers and one is negative, the result may be indeterminate.
Assignment combinations	+ =, etc.	These operators use the value of a memory variable, then change it. Although C permits more than one of these operators to be used in a single expression, you should limit your own expressions to one.
Prefix increment and decrement	++, --	If you use a side-effect operator, do not use the same variable again in the same expression.
Postfix increment and decrement	++, --	Remember that these operators return one value for further use in expression evaluation and leave a different value in memory.
Comparison	==	Remember not to use =.
Logical	&&,   , !	Remember that all negative and positive integers are considered <code>true</code> values. The only <code>false</code> value is 0.
Logical	&&,	There are special sequencing and lazy evaluation rules for expressions that contain these operators.

Figure 4.31. Difficult aspects of C operators.

- Operations for nonsimple types (compound objects with more than one part) will be discussed in Chapters 10, 11, 12, and 13.
- Type conversions (casts and coercions) are explained in Chapter 7.
- Two operators, the question mark and the comma, are not needed in simple programs and are explained in Appendix D. Both are sequencing operators, that is, they have associated sequence points that force left-to-right evaluation.
- More information about evaluation order is given in Appendix D
- When post-increment or post-decrement is used to modify a variable, the time at which the variable is actually changed may vary from compiler to compiler. The C standard permits this variation, within limits, to enable code optimization. The exact rules for when the side effect must and may happen are complex. To explain them, one must first define sequence points and how they are used during evaluation. Consult a standard reference manual for a full explanation.

## 4.10 Exercises

### 4.10.1 Self-Test Exercises

- Look at the parse tree in Figure 4.4. Make a list that shows each operator (one per line) with the left and right operands of that operator.

2. Write a single C expression to compute each of the following formulas:

- (a) Metric unit conversion: Liters = ounces / 33.81474
- (b) Circle: Circumference =  $2\pi r$
- (c) Right triangle: Area =  $\frac{bh}{2}$

3. Each of the following items gives two expressions that are alike except that one has parentheses and the other does not. You must determine whether the parentheses are optional. For each pair, draw the two parse trees and compare them. If the parse trees are the same, the two expressions mean the same thing and the parentheses are optional.

- (a)  $d = a - c + b ; \quad d = a - (c + b) ;$
- (b)  $e = g * f + h ; \quad e = g * (f + h) ;$
- (c)  $d = a + b * c ; \quad d = a + (b * c) ;$
- (d)  $e = f - g - h ; \quad e = (f - g) - h ;$
- (e)  $d = a < b \&& b < c ; \quad d = a < (b \&& b) < c ;$

4. Using the following data values, evaluate each expression and say what will be stored in **d** or **e**:

```
int d, a = 5, b = 4, c = 32;
double e, f = 2.0, g = 27.0, h = 2.5;
```

- (a)  $d = a + c - b ;$
- (b)  $d = a + c * b ;$
- (c)  $d = a * c - b ;$
- (d)  $e = g * 3.0 * (-f * h) ;$
- (e)  $d = a \leq b ;$
- (f)  $e = f - (g - h) ;$
- (g)  $e = g / f ;$
- (h)  $e = 1.0; e += h ;$
- (i)  $d = (a < c) \&& (b == c) ;$
- (j)  $d = ++a * b-- ;$

5. Using the given data values, parse and evaluate each of the following expressions and say what will be stored in **k**. Start with the original values for **k** and **m** each time. Check your precedence table to get the ordering right.

```
double k = 10.0;
double m = 5.0;
```

- (a)  $k *= 3.5;$
- (b)  $k /= m + 1;$
- (c)  $k += 1 / m;$
- (d)  $k -= ++m;$

6. In the following program, circle each error and show how to correct it:

```
#include "stdio"
#define PI 3.14159;
int main (void)
{
    double v;
    printf( "Self-test Exercise/n" );
    printf( "If I don't get going I'll be late!!";
```

```

    puts( "Enter a number:  " );
    scanf( %g, v );
    w = v * Pi;
    printf( "w = %g \n", w );
}

```

7. Draw complete parse trees for the following expressions:

- (a)  $t = x \geq y \ \&\& \ y \geq z ;$
- (b)  $x = (y + z) \ | \ v == 3 \ \&\& \ !(z == y / v) ;$

8. What will be stored in  $k$  by the following sets of assignments? Use these variables: `int h, k, m; .`

- (a)  $h=2; \quad m=3; \quad k = h / m ;$
- (b)  $h=5; \quad m=16; \quad k = h \% m;$
- (c)  $h=10; \quad m=3; \quad k = h / m + h \% m;$
- (d)  $h=17; \quad m=5; \quad k = h / m ;$

#### 4.10.2 Using Pencil and Paper

1. Draw parse trees for the following expressions. Use the trees to evaluate the expressions, given the initial values shown. Assume all variables are type `double`.

- (a)  $a = 5; \ b = 4; \ c = 32; \quad d = a + c / b ;$
- (b)  $w = 3; \ x = 30; \ y = 5; \quad z = y + x / (-w * y) ;$
- (c)  $f = 3; \ g = 30; \ h = 5; \quad d = f - g - h ;$
- (d)  $f = 3; \ g = 27; \ h = 2; \quad d = f - (g - h) ;$

- 2. Explain why you need to know the precedence of the C operators to find the answer to question 1a. Explain why you need to know more than precedence to find the answer to question 1c. What else do you need to know?
- 3. Look at the parse tree in Figure 4.29. Make a list that shows each operator (one per line) with the left and right operands of that operator.
- 4. Parse and evaluate each of the following expressions and say whether the result of the expression is `true` or `false`. Use these variables and initial values: `int h = 0, j = 7, k = 1, n = -3; .`

- (a)  $k \ \&\& \ n$
- (b)  $!k \ \&\& \ j$
- (c)  $k \ | \ j$
- (d)  $k \ | \ !n$
- (e)  $j > h \ \&\& \ j < k$
- (f)  $j > h \ | \ j < k$
- (g)  $j > 0 \ \&\& \ j < h \ | \ j > k$
- (h)  $j < h \ | \ h < k \ \&\& \ j < k$

5. Write a single C expression to compute each of the following formulas:

- (a) Circle: Diameter =  $2r$
- (b) Flat donut: Area =  $\pi \times (\text{outer\_radius}^2 - \text{inner\_radius}^2)$
- (c) Metric unit conversion: cm =  $(\text{feet} \times 12 + \text{inches}) \times 2.54$

6. Parse and evaluate each of the following expressions and say what will be stored in  $k$  and in  $m$ . Start with the original value for  $k$  each time: `int m, k = 10; .`

- (a)  $m = ++k;$

- (b)  $m = k++;$   
 (c)  $m = --k / 2;$   
 (d)  $m = 3 * k --;$
7. (Advanced) Draw parse trees for the following logical expressions and show the sequence points. Use the trees to evaluate the expressions, given the initial values shown. For each one, mark any part of the expression that is skipped because of lazy evaluation.
- |                                   |                      |
|-----------------------------------|----------------------|
| (a) $w = 1; x = 5; y = 1;$        | $y \&& w != y \&& x$ |
| (b) $w = 1; x = 5; y = 3;$        | $w <= x \&& x <= y$  |
| (c) $x = 3; y = 0; z = 0;$        | $z != 0    y \&& !x$ |
| (d) $r = 5; w = 0; x = 5; y = 0;$ | $y    r    x \&& !w$ |
8. What will be stored in  $k$  by the following sets of assignments? All variables are integers.
- |                                      |
|--------------------------------------|
| (a) $h=4; m=5; k = h \% m;$          |
| (b) $h=14; m=7; k = h \% m;$         |
| (c) $h=7; m=15; k = h / m;$          |
| (d) $h=7; m=-5; k = h / m;$          |
| (e) $h=11; m=5; k = h / m + h \% m;$ |
9. (Advanced) Trace the execution of the following loop and show the actual output:

```
int num = 10;
while ( num > 5 ) {
    if ( num % 3 == 0 ) num -= num / 3;
    else if ( num % 3 == 1 ) num += 2;
    else if ( num % 3 == 2 ) num /= 3;
    else num--;
    printf( "num = %i\n", num );
}
```

### 4.10.3 Using the Computer

1. Your own size.

Write a short program in which you use the `sizeof` operator to find the number of bytes used by your C compiler to store values of types `int`, `char`, and `double`.

2. Miles per gallon.

Write a program to compute the gas consumption (miles/gallon) for your car if you are given, as input, `miles`, the number of miles since the last fill-up, and `gals`, the number of gallons of gas you just bought. Start with a formal specification for the program, including a test plan. What is the appropriate type for `miles`? For `gals`? For the answer? Explain why.

3. Centimeters.

Write a program to convert a measurement in centimeters to inches. The numbers of centimeters should be read as input. Define int variable for centimeters, inches, and feet. Convert the centimeters to inches, then convert the inches to feet and inches (use the `%` operator). Print the distance in all three units. There are 2.54 centimeters in each inch and 12 inches in each foot.

Start with a formal specification for the program, including a test plan. Turn in the source code and the output of your program when run using the data from your test plan.

4. A buggy mess.

In the following program, circle each error and show how to correct it. There are errors on nearly every line, totaling at least 15 syntax errors, 4 syntax warnings, 1 linking error, and 3 serious logic errors. When you have found as many errors as you can, download the code from the text website, correct the errors, and try to compile the program. You have successfully debugged this code when you can get the code to compile, run, give you three different multiplication problems, and correctly tell you whether the answers are right or wrong.

```
#include stdio.h
#define SECRET = 17;

int main( void )
{
    integer number wanted;      // The number of problems you want
    integer answer;            // Your answer to the problem

    printf( " Doing the Exercises \n );
           " How many exercises do you want to do: " );
    scanf( "%i", number wanted );
    while ( number wanted > 0 );
    {
        printf( " What is %i * %i? ", SECRET, number wanted );
        scanf( "%i", answer );
        if ( answer = SECRET * number wanted) puts( "Great work." );
        else puts( "You need a calculator!" }
    }
    print( " Thank you for playing today.\n" );
    return;
}
```

5. Turf.

You are a building contractor. As part of a project, you must install artificial turf on some sports fields and the adjacent areas. The owner has supplied length and width measurements of the field in yards and inches. Your supplier sells turf in 1-meter-wide strips that are 4 meters long. Write a program that will prompt the user for a pair of measurements in yards and inches (use integers). Convert each to meters and print the answer. (There are 39.37008 inches in a meter.) Calculate the number of strips of turf needed to cover the field. Round upward if a partial strip is needed.

Start with a formal specification for the program, including a diagram and a test plan. Turn in the source code and the output of your program when run using the data from your test plan.

6. Holy, holy, holy day.

A professor will assign homework today and does not want it due on anybody's holy day. The professor enters today's day of the week (0 for Sunday, 1 for Monday, etc.) and the number of days,  $D$ , to allow the students to do the work, which may be several weeks. Using the % operator, calculate the day of the week on which the work would be due. If that day is someone's holy day—Friday (Moslems), Saturday (Jews), or Sunday (Christians)—add enough days to  $D$  to reach the following Monday. Print the corrected value of  $D$  and the day of the week the work is due.

7. Ascending order.

Write a program to input four integers and print them out in ascending numeric order. Use logical operators when you test the relationships among the numbers.

8. A piece of cake.

Write a complete specification for a program to calculate the total volume of batter needed to half fill two layer-cake pans. The diameter of the pans is  $N$  and they are 2 inches deep. Read  $N$  as an input.

Write a test plan for this program, then write the program and test it. The formula for the volume of a pan is

$$\text{Volume} = \pi \times \frac{\text{diameter}^2}{4.0} \times \text{height}$$

9. Circles.

Write a problem specification and a complete test plan for a program that calculates facts about circles. Prompt the user to enter the diameter of the circle. If the input is less than 0.0, print an error comment. Otherwise, calculate and print the radius, circumference, and area of the circle. Make your output attractive and easy to read, and check it using your test plan.

10. Slope.

The slope of a line in a two-dimensional plane is a measure of how steeply the line goes up or down. This can be calculated from any two points on the line, say  $p_1 = (x_1, y_1)$  and  $p_2 = (x_2, y_2)$  such that  $x_1 < x_2$ , as follows:

$$\text{Slope} = \frac{y_2 - y_1}{x_2 - x_1}$$

Write a specification and test plan for this problem. Then write a program that will input two coordinates for each of two points, validate the second  $x$  coordinate, and print out the slope of the line.

11. What's the difference?

Each term of an arithmetic series is a constant amount greater than the term before it. Suppose the first term in a series is  $a$  and the difference between two adjacent terms is  $d$ . Then the  $k$ th term is  $a + (k - 1) \times d$ . The sum of the first  $k$  terms is

$$\text{Sum} = \frac{k}{2} \times (2a + d \times (k - 1))$$

Write a program that prompts the user for the first two terms of a series and the desired number of terms,  $k$ , to calculate. From these, calculate  $d$  and the sum of the first  $k$  terms. Display  $a$ ,  $d$ ,  $k$ , and the sum.

12. Summing squares.

The sum of the squares of the first  $k$  positive integers is

$$1 + 4 + 9 + \dots + k^2 = \frac{k \times (k + 1) \times (2k + 1)}{6}$$

Write a program that prompts the user for  $k$  and prints out the sum of the first  $k$  squares. Make sure to validate the value of  $k$  that is entered.

13. Greatest common divisor.

Some applications call for performing arithmetic on rational numbers (fractions). To do rational addition or subtraction, one must first convert the two operands to have a common denominator. When doing multiplication or division with fractions, it is important to reduce the result to lowest terms. For both processes, we must compute the greatest common divisor (GCD) of two integers. A good algorithm for finding the GCD was developed by Euclid 2300 years ago. In Euclid's method, you start with the two numbers,  $X$  and  $Y$ , for which you want the GCD. It does not matter which number is greater. Set  $x = X$  and  $y = Y$ , then perform the following iterative algorithm:

- (a) Let  $r = x \% y$ .
- (b) Now set  $x=y$  and  $y=r$ .
- (c) Repeat steps (a) and (b) until  $y == 0$ .
- (d) At that time,  $x$  is the GCD of  $X$  and  $Y$ .

Write a program that will input two numbers from the user and calculate and print their greatest common divisor.



## Chapter 5

# Using Functions and Libraries

In this chapter, we introduce the most important tool C provides for writing manageable, debuggable programs: the function. In modern programming practice, programs are written as a collection of functions connected through well-defined interfaces. We show how to use standard library functions, functions from a personal library, and the programmer's own (programmer-defined) functions.

Functions are important because they provide a way to modularize code so that a large complex program can be written by combining many smaller parts. A **function** is a named block of code that performs a specified task when called. Many functions require one or more arguments. Each **argument** is an object or piece of information that the function can use while carrying out its specified task. Four functions were introduced in Chapter 3: `puts()`, `printf()`, `scanf()`, and `main()`. The first two perform an output task, the third performs input, while the fourth exists in every program and indicates where to begin execution.

Building a program is like building a computer. Today's computer is built by connecting boards. Each board is a group of connected chips, which consist of an integrated group of circuit components constructed by connecting logic elements.

A large program is constructed similarly. At the top level, the program includes several modules, where each module is developed separately (and stored in a separate file.) Each module is composed of object declarations and functions. These functions, in turn, call other functions.

In a well-designed program, the purpose, or task, of each function is clear and easy to describe. All its actions hang together and work at the same level of detail. No function is very long or very complex; each is short enough to comprehend in its entirety. Complexity is avoided by creating and calling other functions to do subtasks. In this way, a highly complex job can be broken into short units that interact with each other in controlled ways. This allows the whole program to be constructed and verified much more easily than a similar program written as one massive unit. Each function, then each module, is developed and debugged before it is inserted into the final, large program. This is how professional programmers have been able to develop the large, sophisticated systems we use today.

**One function is special.** In C, the main program is a special function. In most ways, it is like any function, but `main()` is different in three significant ways:

- Every program must have a `main()` function.
- `main()` is the only function with two standard prototypes:

```
int main( void ); // appropriate for simple programs
int main( int argc, char* argv[] ); // used by some advanced programs
```

- Both prototypes of `main()` are known to the compiler; they do not need to be declared.

## 5.1 Libraries

We use functions from a variety of sources. Many come to us as part of a library, which is a collection of related functions that can be incorporated into your own code. The **standard libraries** are defined by the C language

standard and are part of every C compiler. In addition, software manufacturers often create proprietary libraries that are distributed with the C translator and provide facilities not covered by the standard. Often, a group of computer users shares a collection of functions that become a **personal library**. An individual programmer might use functions from all of these sources and normally also defines his or her own functions that are tailored to the tasks of a particular program.

Using library functions lets a programmer take advantage of the skill and knowledge of experts. In modern programming practice, code libraries are used extensively to increase the reliability and quality of programs and decrease the time it takes to write them. A good programmer does not reinvent the wheel.

### 5.1.1 Standard Libraries

C has about a dozen standard libraries; we use six of them in this text. The first one encountered by a beginning C programmer is the standard input/output library (**stdio**), which contains the functions **scanf()** and **printf()** that we have been using since Chapter 3. This library also contains functions for the input and output of specific data types, which will be explained as those types are introduced, as well as functions for file handling, which will be explained in Chapter 14.

The second most commonly used library is the mathematics library (**math**). This library contains implementations of mathematical functions such as **sin()**, **cos()** and **log**. The program examples in this chapter illustrate the use of several of these functions; a complete list is given in Figure 5.6.

Another important library is the standard library (**stdlib**). It contains functions for generating random numbers; a definition for **abs()**, the absolute value function for integers; and a variety of general utility functions. Several of these will be introduced as the need arises in later chapters. Other libraries that we use are the time library (**time**), the string library (**string**), and the character-handling library (**ctype**).

### 5.1.2 Other Libraries

The standard C libraries contain many useful functions for input and output, string handling, mathematical computations, and systems programming tasks. These libraries provide expert solutions for common needs, but they cannot cover every possible need. The standard libraries contain only a fraction of the useful functions that might be written. The library functions are general-purpose utilities; many were designed for the convenience of programmers creating the **UNIX** operating system. They are not tailored to the needs of students or scientists and engineers who write C programs in the course of their work.

Many C implementations have additional libraries; for example, a graphics library for building screen displays. Special-purpose libraries are often included with hardware that will be connected to a computer. For example, a mobile-robotics kit such as **Lego Mindstorm** includes a library of functions that are used to communicate with the robotics hardware. Finally, many companies that create software have libraries of code relating to their products; by sharing these libraries, employees can become more efficient and products become more uniform and predictable.

Programmers define their own functions to meet their needs. Some are special-purpose functions written for one application and not relevant to other jobs. However, every programmer builds a collection of function definitions that are useful again and again. These usually are simple functions that save a little writing, simplify a repetitive task, or make a programmer's job easier. Often, such a collection is shared with coworkers and becomes a personal library. In this chapter, we suggest several functions that you may wish to put into your own personal library because they will be useful again and again. We call this library "mytools"; it is discussed in Section 5.9.2.

### 5.1.3 Using Libraries

**Prototypes** Every data object has a type. The type of a literal is evident from its form; the type of a variable is declared along with the variable name. Similarly, every function has a type, called a **prototype**, which must be known before the function can be used. The prototype defines the function's **interface**; that is, how it is supposed to interact with other functions. It declares the number and types of arguments that must be provided in every call and the kind of answer that the function computes and returns (if any). This information allows the compiler to check for syntax errors in the function calls.

Nine tenths of an iceberg floats under the surface of the water; we see only a small part on top. Similarly, each C library has two parts: a small public header file that declares the interface for the library and a large, concealed code file that supports the public interface.

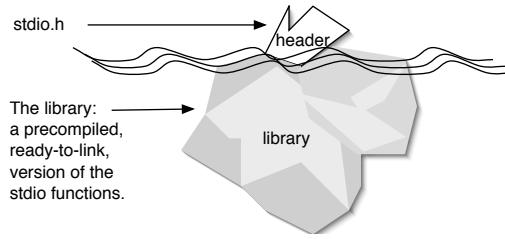


Figure 5.1. A library is like an iceberg: only a small part is visible.

**Header files.** Each standard library has a corresponding header file whose name ends in .h. The **header** file for a library contains the prototype declarations for all of the functions in that library. It may also define related types and constants. For example, the header `time.h` defines a type named `time_t`, which can be used to store the current date and time. This type is related to type `int`, and is chosen to be appropriate for the local computer hardware and software. A useful constant, `INT_MAX` (the largest representable integer) is defined in `limits.h`. Also, the mathematical constant `PI` is defined in `math.h` by many C implementations.

The header files for the libraries we have mentioned so far are

Standard input/output library: < <code>stdio.h</code> >	Standard library: < <code>stdlib.h</code> >
Mathematics library: < <code>math.h</code> >	Time library: < <code>time.h</code> >
Character handling: < <code>ctype.h</code> >	String library: < <code>string.h</code> >
Biggest and smallest integers < <code>limits.h</code> >	Personal library: "mytools.h"

To use one of the library functions in a program, you must include the corresponding header file in your program. This can be done explicitly, by writing an `#include` command for that library, or you can make your own header file that includes the header files for the standard libraries that will be used. Suppose you have such a file called `mytools.h`, that includes `stdio.h`. Then if you write the command `#include "mytools.h"` in your program, there is no need to write `#include <stdio.h>` separately.

In an `#include` command, angle brackets <...> around the name of the header file indicate that the header and its corresponding library are installed in the compiler's standard library area on the hard disk<sup>1</sup>. Use quotation marks instead of angle brackets for personal libraries like `mytools` that are stored in the programmer's own disk directory rather than in the standard system directory.

<sup>1</sup>This file must be stored where your compiler can find it. It always works to put it in the same directory as your program code, or in any directory that is on the compiler's "search path". To find out about the search path, ask your system administrator to help you.

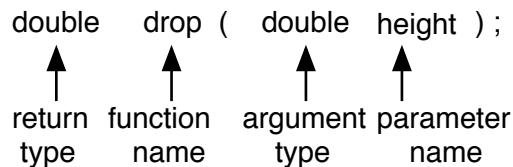


Figure 5.2. Form of a simple function prototype.

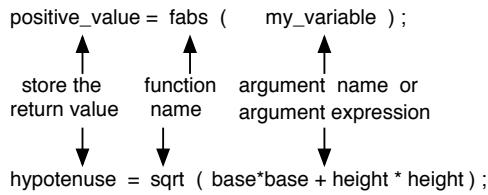


Figure 5.3. Form of a simple function call.

## 5.2 Function Calls

A **function call** causes the function's code to be executed. The normal sequential **flow of control** is **interrupted**, and control is transferred to the beginning of the function. At the end of the function, control returns to the point of interruption. To call a function, write the name of the function followed by a pair of parentheses enclosing a list of zero or more **arguments**. In the following discussion, we refer to the function that contains the call as the **caller** and to the called function as the **subprogram** or, if there is no ambiguity, simply the **function**. Copies of the argument values are made by the caller and sent to the subprogram, which uses these arguments in its calculations. At the end of function execution, control returns to the caller; a **function result** also may be returned.

The function call must supply an argument value for each parameter defined by the function's prototype. The form of a simple prototype declaration is shown in Figure 5.2. It starts with the type of the answer returned by the function. This is followed by the function name and a pair of parentheses. Within the parentheses are zero or more **parameter declaration** units, consisting of a type and an identifier. The type tells us what kind of argument is expected whenever the function is called. The identifier is optional in a prototype (but required in a function definition).

The C compiler checks every function call to ensure that the correct number of arguments has been provided and that every argument is an appropriate type for the function, according to the function's prototype. It also checks that the function's result is used in an appropriate context. Generally, if a mismatch is found between the function's prototype and the function call, the compiler generates an error comment and does not produce a translated version of the code. Some type mismatches are legal according to the type rules of C but they may not be meaningful in the context of the program. In such cases, the compiler generates a warning comment, continues the translation, and produces an executable program. However, the programmer should never ignore warnings; most warnings are clues about logic errors in the program.

**Calling library functions.** The program example in Figure 5.4 demonstrates how to include the library header files in your code and how to call the library functions. It uses standard I/O functions, the `sqrt()` function from the mathematics library, and a function from the `stdlib` library to abort execution after an input error.

### Notes on Figure 5.4. Calling library functions.

#### *First box: the #include commands.*

- We `#include <stdio.h>` so that we can use `printf()` and `scanf()`.
- We `#include <math.h>` for `sqrt()`, the square root function,
- We `#include <stdlib.h>` for the `exit()` function, which is discussed in Section 5.2.3.

#### *Second box: calling exit().*

- The function `exit()` is defined in `stdlib`; its prototype is `void exit( int );`. It can be used to exit from the middle of a program after an error that makes continuation meaningless.
- The symbol `EXIT_FAILURE` is defined as 1. We use the symbolic name here, rather than a literal 1, as a form of program documentation.

- We use a simple `if` statement to test for an input error. If one is found, we display an error message, then call `exit(EXIT_FAILURE)` or `exit( 1 )`. When control returns to the system, it will display a termination comment with the exit code that written in the call on `exit()`.

```
Grapefruits and Gravity with Functions

Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.
Enter height of helicopter (meters): -2
Error: height must be >= 0. You entered -2

Grapefruits has exited with status 1.
```

- No `else` statement is needed because `exit()` takes control immediately and directly to the end of the program. In the flow diagram (Figure 5.17), it is diagrammed as a bolt of lightning because it “short-circuits” all the normal control structures.

*Third box: calling sqrt().*

---

```
// -----
// Grapefruits and Gravity again, with terminal velocity, using sqrt().
//

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

#define GRAVITY 9.8

int main( void )
{
    double height;                      // height of fall (m)
    double time;                        // time of fall (s)
    double velo;                         // terminal velocity (m/s)

    printf( " Grapefruits and Gravity with Functions\n\n"
            " Calculate the time it would take for a grapefruit\n"
            " to fall from a helicopter at a given height.\n"
            " Enter height of helicopter (meters): " );
    scanf( "%lg", &height );             // keyboard input for height

    if (height < 0) {                  // exit gracefully after error
        printf( " Error: height must be >= 0. You entered %g\n", height );
        exit( EXIT_FAILURE );          // abort execution
    }

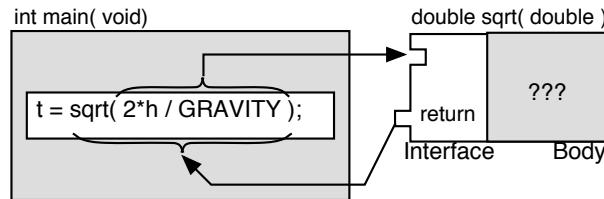
    time = sqrt( 2 * height / GRAVITY ); // calculate the time of fall
    velo = GRAVITY*time;                // terminal velocity of fruit
    printf( "      Time of fall = %g seconds\n", time );
    printf( "      Velocity of the object = %g m/s\n", velo );

    return EXIT_SUCCESS;
}
```

---

Figure 5.4. Calling library functions.

- The `sqrt()` function computes the square root of its argument. To call this function, we write the argument in parentheses after the function name. In this call, the argument is the value of the expression `2*h/GRAVITY`. The multiplication and division will be done, and the result will be sent to the `sqrt()` function and become the value of the `sqrt()`'s parameter. Then the square root will be calculated and returned to the caller. When a function returns a result, the caller must do something with it. In this case, it is stored in the variable named `t`, then used in the next two lines of `main()`.
- A function prototype describes the function's interface; that is, the argument(s) that must be brought into the function and the type of result that is sent back. In this case, one argument is brought in and one result is returned. We can diagram the passage of information to and from the function like this:



- In general, spacing makes no difference to the compiler. We could have written

```
time=by the standard to be an ( 2*height/GRAVITY); or
time =sqrt (2 *height / GRAVITY); or
time= sqrt (2* height/GRAVITY);
```

However, spacing makes an important difference in the readability of a program. You should use spacing selectively to make formulas as readable as possible. Current style guidelines call for spaces after the opening parenthesis and before the closing parenthesis.

*The output from a successful run.*

```
Grapefruits and Gravity with Functions

Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.
Enter height of helicopter (meters): 30
Time of fall = 2.47436 seconds
Velocity of the object = 24.2487 m/s

Grapefruits has exited with status 0.
```

*Fourth box: the return statement.*

- In previous programs, we have written `return 0;`. Here we introduce another way to write the same thing: `return EXIT_SUCCESS`. The `stdlib.h` header file defines `EXIT_SUCCESS` to be a synonym for 0, and some programmers prefer to use the symbolic name rather than the numeric value.
- The last line of the output, above, shows the message printed by the operating system after the program terminated. The status code that is displayed is the value that was written in the return statement.

### 5.2.1 Call Graphs

We use flow diagrams (as in Figure 5.17) to visualize the flow of control through the statements of a single function or the transition from one function to another during execution. Another kind of diagram, a **function call graph**, is useful for showing the relationships between functions that are established by function calls. In a function call graph (see Figure 5.5), a box at the top is used to represent the function `main()`. Below it, attached to the branches of a bracket, is one box for each of the functions called by the main program.<sup>2</sup> As far as possible, these are listed left to right in the order in which they appear in the program. Each function has only one box; if it is called several times, there is no sign of that in the diagram. A very simple program is graphed in Figure 5.5.

<sup>2</sup>Various elaborations of this basic scheme are in use. In one version, the function arguments are written on the arrows. We choose to introduce the concepts by using the simplest scheme.

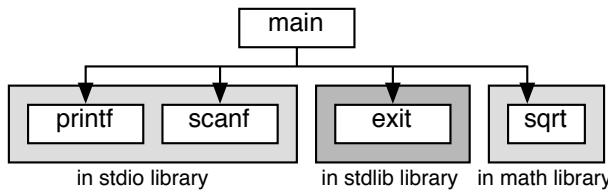


Figure 5.5. Call graph for Figure 5.4: Calling library functions.

The pattern of one box pointing at others is repeated when diagramming a more complex program; a box and a connection are drawn for each function called by a first-level function. In Figure 5.12, the main program calls a programmer-defined function, `banner()` (Figure 5.13), which in turn, calls functions named `time()` and `ctime()` from the `time` library. The resulting function call graph, shown in Figure 5.14, has boxes at three levels. In a large program, a function call graph will have many boxes at several levels and may have complex dependencies among the functions (arrows may point from a lower level to an upper level). We will use call graphs to visualize the relationships among functions in future program examples. The graph becomes more and more important as the number of functions increases and the interactions among functions become more complex.

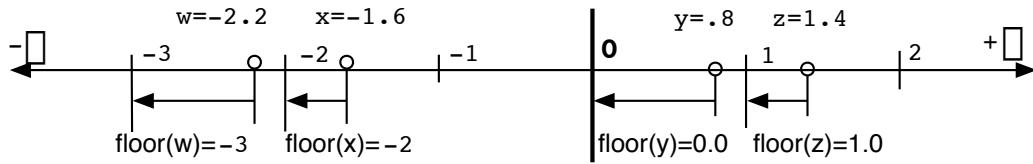
In some programs it becomes beneficial to add information to the call graph concerning what is being passed between the functions. This would require an arrow for every parameter in every function call. If a program makes many function calls, this can get very complicated very quickly. Therefore, for the sake of clarity, we omit parameter and return information from the call graphs presented in this text.

### 5.2.2 Math Library Functions

Most of the functions in the math library are familiar to anyone who has studied high-school algebra and trigonometry. These include the trigonometric, exponential, log, and square root functions. Some of the

Name	Function	Argument type(s)	Return type
<code>fabs(x)</code>	Absolute value	double	double
<code>ceil(x)</code>	Round $x$ up	double	double
<code>floor(x)</code>	Round $x$ down	double	double
<code>rint(x)</code>	Round $x$ to nearest integer	double	double
<code>cos(x)</code>	Cosine of $x$	double	double
<code>sin(x)</code>	Sine of $x$	double	double
<code>tan(x)</code>	Tangent of $x$	double	double
<code>acos(x)</code>	Arc cosine of $x$	double	double
<code>asin(x)</code>	Arc sine of $x$	double	double
<code>atan(x)</code>	Arc tangent of $x$	double	double
<code>atan2(y, x)</code>	Arc tangent of $y/x$	double, double	double
<code>cosh(x)</code>	Hyperbolic cosine of $x$	double	double
<code>sinh(x)</code>	Hyperbolic sine of $x$	double	double
<code>tanh(x)</code>	Hyperbolic tangent of $x$	double	double
<code>exp(x)</code>	$e^x$	double	double
<code>log(x)</code>	Natural log of $x$	double	double
<code>log10(x)</code>	Base 10 log of $x$	double	double
<code>sqrt(x)</code>	Square root	double	double
<code>pow(x, y)</code>	$x^y$	double, double	double
<code>fmod(x, y)</code>	$x - N \times y$ for largest $N$ such that $N \times y < x$	double, double	double

Figure 5.6. Functions in the math library.

Figure 5.7. Rounding down: `floor()`.

math library functions are less familiar; these are briefly explained in the next several paragraphs, with a few important functions from other libraries.

**int abs( int n ).** Although this is a mathematical function, it is part of the standard library (`stdlib`), not the math library; to use it, a program must `#include <stdlib.h>`. The result is the absolute value of  $n$ . (That is, if  $n$  is negative, the result has the same value as the argument, but with a positive sign instead of a negative sign. If the argument is zero or positive, the result is the same as the argument.)

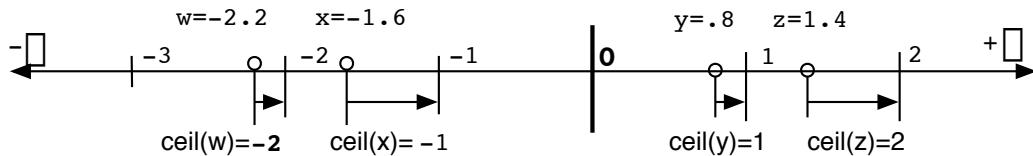
**double fabs( double x ).** To use this function, `#include <math.h>`. This is just like `abs()` but it works for a `double` argument instead of an `int`, and it returns a `double` result. The result is the absolute value of  $x$ . (That is, if  $x$  is negative, the result has the same value as the argument, but with a positive sign instead of a negative sign. If the argument is zero or positive, the result is the same as the argument.)

**double fmod( double x, double y ).** To use this function, `#include <math.h>`. C has an operator, `%`, that computes the modulus function on integers; `fmod()` computes a related function for two `doubles`. The result of `fmod(x, y)` is the floating-point remainder of  $x/y$ . More precisely,  $fmod(x, y) = x - k*fabs(y)$  for some integer value  $k$ . The result has the same sign as  $x$  and is less than the absolute value of  $y$ . The function is implementation-defined if  $y==0$ . A few examples might make this clear:

```
fmod( 10.0, 2.0 ) = 0      and k = 5
fmod( -10.0, 2.9 ) = -1.3  and k = 3
fmod( 10.5, -1.0 ) = .5    and k = 10
fmod( 10.5, 1.1 ) = .6    and k = 9
fmod( 34.5678, .01 ) = 0.0078
```

The last example, above, hints at an application for `fmod()`. Suppose a bank calculates the amount of interest due on an account, but will add only an even number of cents to the account balance. The fractional cents must be subtracted from the calculated interest before adding the interest to the account balance. This function lets us easily calculate the fractional cents.

**double atan2( double x, double y ).** This function computes the arc tangent of  $x/y$ . It is explicitly defined for  $y=0$  is  $\pi/2$  and has the same sign as  $x$ . This should be used in place of `atan()` for any argument expression that might have a denominator of 0.

Figure 5.8. Rounding up: `ceil()`.

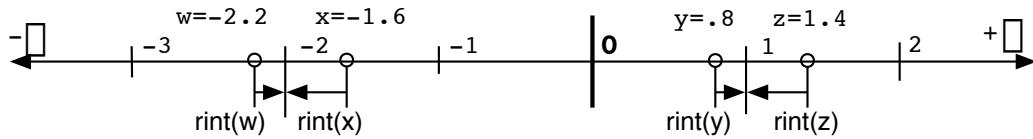


Figure 5.9. Rounding to the nearest integer: `rint()`.

**Rounding and truncation.** Suppose that we are given a `double` value and wish to eliminate the fractional part. The math library provides three functions that correspond to three of basic ways to do this job. The assignment operator provides a fourth way.

- The function `floor()` rounds down, that is, the result will be an integral value that is closer to  $-\infty$  than the argument, as illustrated in Figure 5.7. This function returns a `double` value. For example, `floor(-1.6)` is `-2.0` and `floor(.9999999)` is `0.0`.
- The function `ceil()` (short for ceiling) rounds up, that is, the result will be an integral value that is closer to  $+\infty$  than the argument, as illustrated in Figure 5.8. This function returns a `double` value.
- The function `rint()` does what we normally refer to as “rounding”. It rounds to the nearest integral value and returns it as a `double` value, as illustrated in Figure 5.9. A matching function, `lrint()`, returns the result as an integer, if it is possible to represent it as an integer. For example, `rint(-1.6)` is `-2.0` and `rint(.9999999)` is `1.0`. Compare this result to `lrint(.9999999) = 1`
- When a double value is assigned to an integer variable, the fractional part is *truncated*, that is, the decimal places are simply discarded. This is the same as rounding positive numbers down (toward 0) and rounding negative numbers up (toward 0).

### 5.2.3 Other Important Library Functions

**The date and time.** The date and time at which a program is executed is very important for many kinds of output, including student work. Modern computers have an internal battery-operated clock. Modern systems keep that clock accurate by periodically synchronizing it to a time standard accessed over the internet. C provides a variety of functions and type definitions to help programmers use the clock<sup>3</sup>. Most of the time library is too difficult for this chapter, but a few items can be used simply and are presented here.

- The data type `time_t` is defined<sup>4</sup> by the standard to be an integer that has enough bits to hold whatever encoding of date and time is used by the local system. It might be different from system to system, but it is always exactly right to store the time. The program in Figure 5.13 shows how to declare a variable of this type and use it to store the current time.
- The `time( )` function reads the system clock and returns an integer encoding of the time and date. It is described in Appendix F and discussed in more detail in Chapter 12. Until then, if you want to read the system clock, you should call the `time( NULL )` and store the result in a `time_t` variable.
- The `ctime( )` function is used to convert the time from the coded form to a string that can be easily printed and understood. The easiest way to use `ctime( )` is to call the function from the argument list of

<sup>3</sup>These are described in Appendix F and parts are discussed in detail in Chapter 12.

<sup>4</sup>Until now, we have covered only three types: `double`, `int`, and `char`. The C language actually supports many predefined types and permits the programmer to define his own. These will be introduced gradually in future chapters.

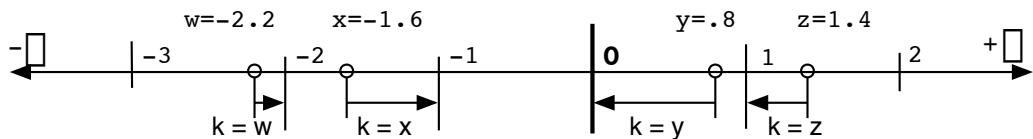


Figure 5.10. Truncation via assignment.

Library	Purpose	Name and Usage	Argument	Return type
stdlib	Absolute value	<code>k = abs( amount );</code>	int	int
stdlib	Abort execution after an error	<code>exit(1)</code>	int	returns to system
time	Data type for storing the time	<code>time_t now;</code>		creates a time variable
time	Read system clock	<code>now = time( NULL )</code>	NULL	the time encoded as an integer
time	Convert time code for printing	<code>printf( ctime( &amp;now ) )</code>	integer	a printable string

Figure 5.11. A few functions in other libraries.

a `printf()` function. The argument to `ctime( )` must be the address of the `time_t` variable that was set by calling `time(NULL)`. Figure 5.13 shows how to do this.

**Exception conditions.** If a program encounters a serious error and cannot meaningfully continue, the appropriate action is to stop execution immediately. C++ and Java are newer languages that are built upon C; both include all the C operators and control structures, but provide an additional modern facility, called an exception handler, for managing errors and other unusual and unexpected conditions. C is an old language and it does not provide an exception handler. However, it does provide a function that aborts a program and “cleans up” the environment before returning to the system. In this text, the `exit()` function will be used when it would be appropriate to use an exception in a modern language.

- The function `exit()` is defined in `stdlib`; its prototype is `void exit( int );`. It can be used to exit from the middle of a program after an error that makes continuation meaningless.
- The constant `EXIT_FAILURE` is defined in `stdlib.h` as a synonym for the number 1 and the constant `EXIT_SUCCESS` is defined as a synonym for the number 0. These can be used as arguments to `exit()`, but a programmer can also use any integer as the argument or invent his or her own codes. The system should display the code on the screen after the program exits.

### 5.3 Programmer-Defined Functions

Functions serve three purposes in a program: They make it easy to use code written by someone else; they make it possible to reuse your own code in a new context; most important, though, they permit breaking a large program into small pieces in such a way that the interface between pieces is fixed and controllable. A programmer may (and generally does) modularize his or her program by dividing the entire job into smaller tasks and writing a **programmer-defined function** for each task.

Functions can take no arguments or many, of any combination of types, and can return or not return values. The type of a function is a composite of the type of value it returns and the set of types of its arguments.

Function	Prototype
<code>main()</code>	<code>int main( void );</code>
<code>exit()</code>	<code>void exit( int );</code>
<code>sqrt()</code>	<code>double sqrt( double );</code>

We begin the study of programmer-defined functions by creating functions of two types: `double`→`double` and `void`→`void`.

**Double→double functions.** Some functions calculate and return values when they are called. For example, in Figure 5.4, the function `sqrt()` calculates a mathematical function and returns the result to `main()`, which stores it in a variable and later prints it. The functions `sqrt()`, `log()`, `sin()`, and `cos()` all accept an argument of type `double` and return an answer of type `double`. We say, informally, that these are **double**→**double** functions because their prototypes are of the form `double funcname(double)`. A `double`→`double` function must be called in some context where a value of type `double` makes sense. Often, these functions are called from the right side of an assignment statement or from a `printf()` statement. Examples of calls on `double`→`double` functions follow:

```
time = sqrt( 2 * height / GRAVITY );
printf( "The natural log of %g is %g\n", x, log( x ) );
```

**Void functions.** Some functions return no value to the calling program. Their purpose is to cause a side effect; that is, perform an input or output operation or change the value of some memory variable. These functions are called *void functions* because their prototypes start with the keyword `void` instead of a return type. `Void` means “nothing”; it is not a type name, but we need it as a placeholder to fill the space a type name normally would occupy in a prototype. We need some keyword because, if the return-type field in a function header is left blank, the return type defaults to `int`.<sup>5</sup>

Some `void` functions, such as `exit()`, require arguments; others, do not. The latter are called **void→void functions** and have prototypes of the form `void funcname( void )`. To call a void→void function, write the function name followed by empty parentheses and a semicolon. Often, the function call will stand by itself on a line.

### 5.3.1 Function syntax.

A function has two parts: a **prototype** and a **definition**. When writing a program, prototypes for all the functions are normally written at the top, between the preprocessor commands and the beginning of `main()`. Function definitions are written at the bottom of the file, following the end of `main()`.

A function definition, in turn, has two parts: a **function header** (which must correspond to the prototype) and a **function body**, which is a block of code enclosed in curly brackets. The body starts with a series of (optional) declarations. These create local variables and constants for use by the function. The local declarations are followed by a series of program statements that use the local variables and the function’s arguments (if any) to compute a value or perform a task. The body may contain one or more `return` statements, which return a value to the calling program.

The complete set of rules for creating and using functions in C is extensive and complex; it is presented in some detail in Chapter 9. In this section, we begin by writing the two types of functions discussed above. We illustrate how to define these function types, write prototypes for them, call them, and draw flowcharts (using barred boxes) to show the flow of control. We give a few examples and some brief guidelines so that the student may begin using functions in his or her own programs. The next figures illustrate, in context, how to write the parts of void→void and double→double functions.

### 5.3.2 Defining Void→Void Functions

We show how to write void→void functions first, using Figure 5.12 to illustrate the discussion. This program uses a void→void function to print user instructions and error comments. We ask the user to enter the number of passengers in a car. If the input number is greater than 5, we print an error message and beep four times.

#### Notes on Figures 5.12 and 5.13. Calling void→void functions.

##### *First box: the standard header files.*

- The prototypes for the time library and the standard I/O library are brought into the program by these `#include` statements.
- The header `<time.h>` is included because one of the programmer-defined functions will call functions from the time library. When we include prototypes at the top of a program, either explicitly or by including a header file, the corresponding functions can be called anywhere in the program.

##### *Second box: the prototypes.*

- Either a prototype declaration or the actual function definition should occur in a program before any calls on the function.<sup>6</sup>
- The include statements bring in prototypes for the standard functions. However, we must write prototypes for the three functions defined at the bottom of this program, `beep()`, `instructions()`, and `banner()`.

---

<sup>5</sup>This default makes no sense in ISO C; it is an unfortunate holdover from pre-ISO days, when C did not even have a type `void`. The default to type `int` was kept in ISO C to maintain compatibility with old versions of C.

<sup>6</sup>If a function is called before it is declared, the C compiler will construct a prototype for that function that may or may not work properly.

- The prototype for every void→void function follows the simple pattern shown here: the word `void` followed by the name of the function, followed by the word `void` again, in parentheses. Every prototype ends in a semicolon.

***Third and fourth boxes: the function calls.***

- These lines all call a void→void function. The call consists of the function name followed by empty parentheses. Often, as with the calls on `instructions()` and `banner()`, a void→void function call will stand alone on a line.
- Often a void→void function is used to give general instructions or feedback to the user, as in the second box. Sometimes, one is used to inform the user that an error has happened, as is the case with the call on `beep()` in the third box. Such calls often form one clause of an `if` statement.

***Fourth and fifth boxes: two easy function definitions.***

- We define two void→void functions: `instructions()` and `beep()`. We use a comment line of dashes to mark the beginning of each function so that it is easy to locate on a video screen or printout.

The banner function is shown in Figure 5.13.

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

// Prototype declarations for the programmer-defined functions. -----
void banner( void );
void instructions( void ) ;
void beep( void );

int main( void )
{
    int n_pass;                                // Number of passengers in the car.

    banner();                                    // Display output headings.
    instructions();                            // Display instructions for the user.

    scanf( "%i", &n_pass );

    if (n_pass > 5) beep();      // Error message for n>5
    else printf( " OK! \n" );    // Success message for good entry
    return 0;
}

// -----
void instructions( void )                  // function definition
{
    printf( " This is a legal-passenger-load tester for 6-seat sedans.\n"
           " Please input the number of passengers you will transport: " );
}

// -----
void beep( void )                          // function definition
{
    printf( " Bad data! \n\aa\aa\aa\aa" );    // error message and beeps
}
```

Figure 5.12. Using void→void functions.

This function is part of the program in Figure 5.12, and belongs at the bottom of the same source-code file.

```
// -----
void banner( void ) // Print a neat header for the output.
{
    time_t now = time(NULL);

    printf( "\n-----\n" );
    printf( "      Patience S. Goodenough\n      CS 110\n      " );
    printf( ctime( &now ) );
    printf(
        "-----\n" );
}
```

**Figure 5.13.** Printing a banner on your output.

- Each function definition starts with a header line that is the same as the prototype except that *it does not end in a semicolon*. Following each header line is a block of code enclosed in curly brackets that defines the actions of the function.
- Most void→void functions, like these two, perform input or output operations. The symbol \a, used in the beep() function, is the ASCII code for a beeping noise. Many systems will emit an audible beep when this symbol is “printed” as part of the output. Other systems may print a small box instead of emitting a sound. Some systems give no audible or visible output.

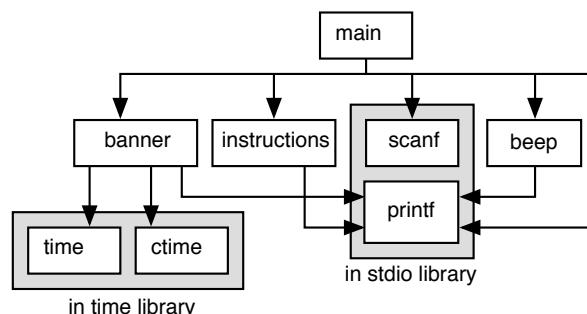
#### **The output**

Here is the output from two test runs (the second banner has been omitted):

```
-----
Patience S. Goodenough
CS 110
Sat Aug  9 18:21:17 2003
-----
This is a legal-passenger-load tester for 6-seat sedans.
Please input the number of passengers you will transport: 2
OK!

Tester has exited with status 0.
-----
This is a legal-passenger-load tester for 6-seat sedans.
```

Library functions are surrounded by shaded boxes; programmer-defined functions are shown with no surrounding box.



**Figure 5.14.** A call graph for the beep program.

```

Please input the number of passengers you will transport: 10
Bad data!

Tester has exited with status 0.

```

**Figure 5.13** a longer function definition.

- We define a void→void function named `banner()` that prints a neat and visible output header containing the programmer's name and the date and time when the program was executed. The date and time are produced by calling functions from the `time` library.
- The first line of this function declares a variable named `now` of type `time_t` and initializes it to the current time by calling `time( NULL )`.
- Then we call `ctime()` to convert the coded date and time into a string that can be printed. The third call on `printf()` shows how do use this function. Be sure to write the ampersand when you copy this code.
- An output header, similar to the one this function produces should be part of the output produced by every student program. You should use it with everything you hand in to the teacher. One way to do this is to start your own personal file, say `mytools.c` containing reusable code. Make a matching file called `mytools.h` containing the prototypes for the functions in `mytools.c`. Include `mytools.h` in every main program you write, and add both `mytools` files to the project you create for your program.

### 5.3.3 Returning Results from a Function

A function that returns a value is fundamentally different from a `void` function. A `void` function simply causes some side effect, such as output, like `banner()` in Figure 5.19. The call on such a function forms a separate statement in the code. In contrast, a function that returns a value interacts with the rest of the program by creating information for further processing. A `return` statement is used to send a result from a function back to the caller. It is represented in the diagram in Figure 5.21 as a tab sticking out of the function's interface. A `return` statement can be placed anywhere in the function definition, and more than one `return` statement can be used in the same function.<sup>7</sup>

A function that returns a value can be called anywhere in a C statement that a variable name or literal of the same type would be permitted. Often, as in the call on `f()` in Figure 5.19, a function is called in an assignment statement. The return address for this call is in the middle of the statement, just before the assignment happens. When the value is returned from the call, `main()` will resume execution by assigning the returned value to the variable `z`.

If a function is called in the middle of an expression, the result of the function comes back to the calling program in that spot and is used to compute the value of the rest of the expression. The call on `exp()` in Figure 5.19 illustrates this. The function is called from the middle of a `return` statement: `return y * exp(y)`. The return address for this call is in the middle of the statement, just before the multiplication happens. After a value is returned by `exp()`, it will be multiplied by the value of `y` and the result returned to `main()`.

Finally, as in the call on `g()` in Figure 5.19, a function can be called from the argument list of another function. It is quite common to nest function calls in this way.

### 5.3.4 Arguments and Parameters

Function parameters introduce variability into the behavior of a function. A void→void function without parameters always does the same thing in the same way.<sup>8</sup> In contrast, introducing even one parameter permits the actions of a function and its results to depend on the data being processed. By parameterizing a piece of code, we can make it useful under a much more general set of circumstances.

**Formal parameters** are part of a function definition and specify a set of unknowns; **arguments** are part of a function call and supply values for those unknowns. In Figure 5.21, parameters are represented by notches along the left edge of each function's interface. Right-facing arrows connect each argument to the notch of the corresponding parameter; these arrows represent the direction in which information flows from the caller to the subprogram.

<sup>7</sup>However, we strongly recommend using a single `return` statement at the end of the function.

<sup>8</sup>An exception to this occurs if the function uses global variables or user input.

The function `f()` has one parameter, a `double` value named `y`. Even if other objects in the program have the same name, the parameter `y` in `f()` will be a distinct object, occupying a separate memory location. It is quite common to have two objects with the same name defined in different functions.

Looking at the list of library functions in Figure 5.6, we see that the `exp()` function has one parameter of type `double`. The name of this parameter is not known because `exp()` is a library function and its details have been concealed from us. During the calling process, the argument value is stored in the parameter variable, making a complete object.

A function can be `void` or have one or more parameters. Its prototype defines the correct **calling sequence**; that is, the number, order, and types of the arguments that must be written in a call. The call must supply one argument expression per parameter;<sup>9</sup> if the number of arguments does not match the number of parameters, the program will not compile. When a function call is executed, each argument expression in the call will be evaluated and its value will be passed from the caller to the subprogram, where it will be stored in the corresponding parameter. For example, the argument in the call on `f()` is the value of the variable named `x` in the main program. This value is a `double`, so it can be stored in the `double` parameter with no conversion. An ISO C prototype states the name of a function, the types of its parameters, and the return type. The parameters also may be named in the prototype, but such names are optional and often omitted. A function header states the same information, except that parameter names *are required* in the function header.

Inside a function, the parameter names are used to refer to the argument values; the first parameter name in the function header refers to the first argument in the function call, and so on. In Figure 5.21, when `main()` makes the call `f(x)`, the value of `x` is copied into the parameter named `y`. Within the body of `f()`, the value stored in this parameter will be used wherever the code refers to the name `y`. During execution of `f()`, this value is further copied and stored in the parameter variable of `exp()`.

**Formal Parameter Names** The function header (which is the first line of the function definition) the name of a function, the types of its parameters, and the return type. These names provide a way for the programmer to refer to the parameters in the function's code.

Any legal name may be given to a formal parameter. It may be the same as or different from the name of a variable used in the function call, and both can be the same as or different from the optional name in the function's prototype.<sup>10</sup> The names chosen do not affect the meaning of the program because argument values are matched up with parameters by position, not by name. For example, the main program in Figure 5.15 uses a variable named `h` in the call on the `drop()` function (Figure 5.18). Within `drop()`, however, the parameter is named `height`, so the value of `main`'s `h` will be stored in `drop`'s `height`.

### 5.3.5 Defining a Double→Double Function

We use Figures 5.15 through 5.18 to illustrate the construction and call of a double→double function, which is somewhat more complicated than a void→void function because argument information must be passed into the function and a result must be returned to the caller.

**Notes on Figure 5.15. The grapefruit returns.**

**First box: the prototypes.**

- A prototype for a double→double function gives the function name and specifies that it requires one `double` parameter and returns a `double` result. The general form is

```
double function_name( double parameter_name );
```

where the parameter name is optional.

- This box contains a prototype declaration for the function named `drop()`. It states that `drop()` requires one argument of type `double` and returns a `double` result. This information permits the C compiler to check whether a call on `drop()` is written correctly.
- There is also a prototype declaration for the void→void function named `title()`, which is similar to the `instructions()` function in the previous example.

---

<sup>9</sup>Some functions accept a variable number of arguments; `scanf()` is an example. However, the details of how this is accomplished are beyond the scope of this text.

<sup>10</sup>However, it is good style to use the same name in the prototype and the function header.

**Second box: the void→void function call.** The prototype for `title()` must precede this function call. The definition of the function is at the bottom of the program, in the fourth box.

A function call interrupts the normal sequence of execution. Look at the flow diagram in Figure 5.17. The first statement is the call on `title()`. When that call is executed, control leaves the main sequence of boxes and travels along the dotted line to the beginning of the `title()` function. Then control proceeds, in sequence, to the return statement, and finally returns to where it came from along the second dotted arrow.

**Third box: the double→double function call.** This box calls `drop()`. The prototype for `drop()` was given in the first box and the function is defined in Figure 5.18, but would be placed in the same source file, below the definition of `title()`.

The form of any function call must follow the form of the prototype. When we call a double→double

A grapefruit is dropped from a helicopter hovering at height `h`. This continues development of the program in Figure 5.4. The `drop()` function is shown in Figure 5.18.

```
// -----
// Modify the Grapefruits and Gravity program by using a double→double
// function to compute the travel time of the grapefruit.
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define GRAVITY 9.81

void title( void );
double drop( double height ); // Prototype declaration of drop.

int main( void )
{
    double h; // height of fall (m)
    double t; // time of fall (s)
    double v; // terminal velocity (m/s)

    title(); // Call function to print titles.

    printf( " Enter height of helicopter (meters): " );
    scanf( "%lg", &h ); // keyboard input for height

    t = drop( h ); // Call drop. Send it the argument h.

    v = GRAVITY * t; // velocity of grapefruit at this time

    printf( "     Time of fall = %g seconds\n", t );
    printf( "     Velocity of the object = %g m/s\n", v );
    return 0;
}

// -----
void title( void ) {
    printf(" Grapefruits and Gravity with a Drop Function\n\n"
          " Calculate the time it would take for a grapefruit\n"
          " to fall from a helicopter at a given height.\n" );
}
```

Figure 5.15. The grapefruit returns.

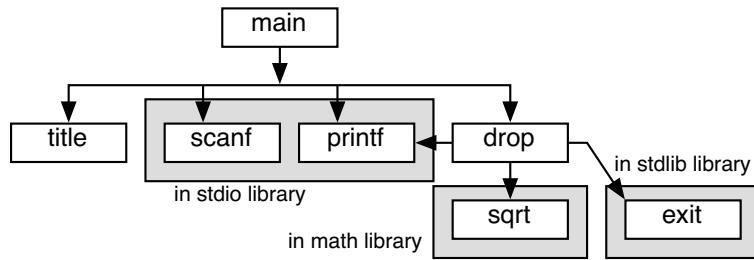


Figure 5.16. Call graph for the grapefruit returns.

function, we must supply one argument expression of type `double`. In this call, the argument expression is a simple variable name, `h`. When we call `drop()`, we send a copy of the value of `h` to the function to be used in its calculations.

A function call interrupts sequential execution and sends control into the function. Figure 5.17 depicts this interruption as a dotted arrow going from the function call to the beginning of the function. From there, control flows through the function to the return statement, which sends control back to the point of interruption, as shown by the lower dotted arrow. When a double→double function returns, it brings back a `double` value, which should be either used or stored. In this example, we store the result in the `double` variable `t`.

**Program output.** Here is one set of output from the grapefruit program (the banners and termination messages have been omitted):

```

Grapefruits and Gravity with a Drop Function
Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.
Enter height of helicopter (meters): 872
Time of fall = 13.3401 seconds
Velocity of the object = 130.733 m/s
  
```

**Notes on Figure 5.18. Definition of the `drop()` function.** We show how to write the definition of a double→double function.

#### The comment block and the function header.

- Every function should start with a block of comment lines, the **function comment block**, that separate the definition from other parts of the program and describe the purpose of the function. Comment marks (`/*...*/`) begin the first line and end the last line of the block comment. These comments provide a neat and visible heading for the function.
- The first line of a function definition is called the *function header*. It must be like the prototype except that the parameter name is required (not optional) in the header and the header does not end with a semicolon.
- A parameter can be given any convenient name, which need not be the same as the name of the argument that will appear in future function calls. A new variable is created for the parameter and can be used only by the function itself.
- When a function is called, the expression in parentheses is evaluated and its value passed into the function and used to initialize the parameter variable. This value is called the *actual argument*. Within the function, the parameter name is used to refer to the argument value. In the function call, the argument was `main()`'s variable `h`, but the `drop()` function's parameter is named `height`. This is no error. *A double→double function can be called with any double argument value.* The argument can be the result of an expression, a variable with the same name, or a variable with a different name. Within the function, the parameter name (`height`) is used to refer to the argument value.

This is a flow diagram of the program in Figures 5.15 and 5.18. Function calls are depicted using a box with an extra bar on the bottom to indicate a **transfer of control**. The dotted lines show how the control sequence is interrupted when the function call sends control to the beginning of the `drop()` function. Control flows through the function then returns via the lower dotted line to the box from which it was called. The call on `fatal()` (after an error) ends execution immediately and returns control to the operating system.

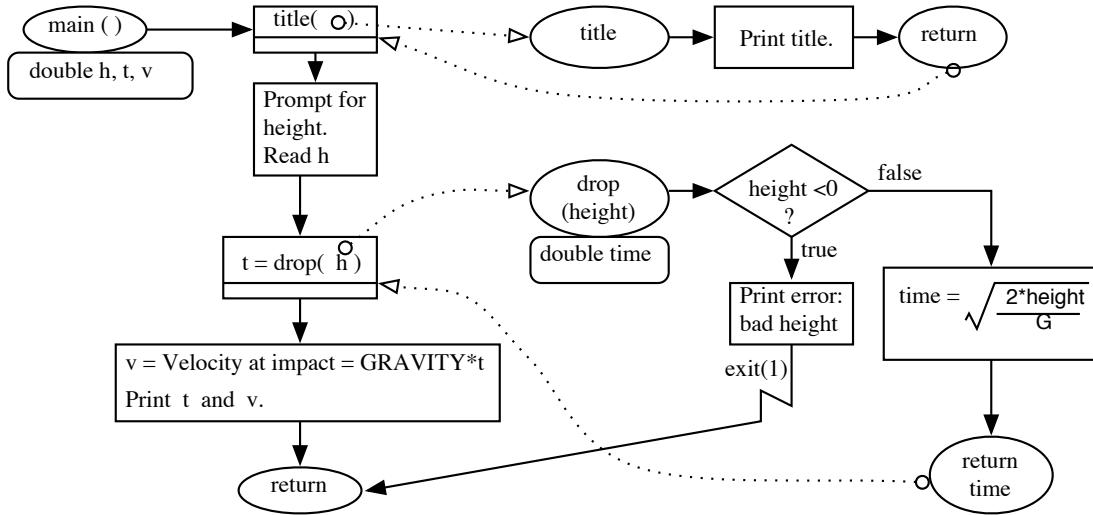


Figure 5.17. Flow diagram for the grapefruit returns.

This function is called from Figure 5.15.

```

// -----
// Time taken for an object dropped from height meters to hit the ground.
double drop( double height )
{
    double time; // create a local variable

    if (height < 0) { // exit gracefully after error.
        printf( " Error: height must be >= 0. You entered %g\n", height );
        exit( 1 ); // abort execution
    }
    time = sqrt( 2 * height / GRAVITY ); // calculate the time of fall

    return time;
}
  
```

Figure 5.18. Definition of the `drop()` function.

**First box: a local variable declaration.** The code block of a function can and usually does start with declarations for **local variables**. Memory for these variables will be allocated when the function is called and deallocated when the function returns. These variables are for use only by the code in the body of the function; no other function can use them. In the example program, we declare a local variable named `time`.

**Second box: the function code.** Statements within the function body may use the parameter variable and any local variables that were declared. References also may be made to constants defined globally (at the top of the program). The use of global variables is legal but strongly discouraged in C. The statements in this function are like the corresponding lines of Figure 5.4 except that they use the parameter name and local variable name instead of the names of the variables in the main program.

**Inner box: calling another function.** An error has been detected, so we print an appropriate comment and call `exit( 1 )` to terminate the program immediately.

**Last box: the return statement.** On completion, a double→double function must return a result of type `double` to its caller. This is done using the `return` statement. A `return` statement does two things: it sends control immediately back to the calling program and it tells what the return value (the result) of the function should be. It does not need parentheses. In Figure 5.18, the result from `sqrt()` is stored in the local variable `time`. To make that answer available to the caller, we return the value of `time`. On executing the `return` statement, the value of `time` is passed back to the caller and control is returned to the caller at the statement containing the function call, as depicted by the dotted line in Figure 5.17.

## 5.4 Organization of a Module

Generally, a program has a `main()` function that calls several other library and programmer-defined functions. To compile `main()`, the compiler needs to know the **prototype** of every function called. One way this information can be supplied is by putting `main()` at the bottom of the module, while the definitions of the other functions come before it. However, many programmers dislike having the main program at the end of the file and it is customary, in C, to put `main()` at the top. When this is done, prototypes for all the functions that `main()` calls must be written above<sup>11</sup> it. This pattern has been followed in every program example given so far. There is one major exception to this organizational guideline: When a function is so simple that its entire definition can fit on one line, the function itself often is written at the top of the file in place of a prototype.

When you call a function from one of the C libraries, you use code that is already compiled and ready to link to your own code (see Chapter 5). Header files such as `stdio.h` and `math.h` contain prototypes (not C source code) for the precompiled library functions. When your code module uses a **library** function you `#include` the library header file at the top. This causes the preprocessor to insert all the prototypes for the library functions into your module, making it possible for the compiler to properly check your calls on the library functions. The order of parts, from the top of the source file to the bottom follows. These principles lead to the following layout for the parts in a simple program:

- `#include` commands for header files.
- Constant definitions and type declarations.<sup>12</sup>
- Prototypes (function declarations) and one-line functions.
- `main()`, which contains function calls.
- Function definitions, possibly containing more calls.
- Figure 5.19 illustrates the principles with a complete program and two functions.

### Notes on Figure 5.19. Functions, prototypes, and calls.

---

<sup>11</sup>The prototypes also may be written inside the calling function. However, we wish to discourage this practice.

<sup>12</sup>Type declarations will be discussed in Chapters 11, 12 and 13.

A function's prototype may be given first, then the call, and finally the full definition of the function, like function `f()` here. Alternatively, the function may be fully defined before it is called; for example, function `g()` is defined before `main()`, which calls it.

```
#include <stdio.h>
#include <math.h>
#define MIN 10.5
#define MAX 87.0

double f( double y );           // Prototype for function f, defined below.
double g( double y ) { return( y * y + y ); } // Definition of function g.

int main( void )
{
    double x, z;
    banner();
    printf( "\n Enter a value of x between %.2f and %.2f: ", MIN, MAX );
    scanf( "%g", &x );
    z = f( x );
    printf( "\n The value of x * exp(x) is: %g \n", z );
    printf( "\n The value of x * x + x is: %g \n", g( x ) );
    return 0;
}

// -----
// Definition of function to calculate f = y * e to the power y. -----
double
f( double y )
{
    return y * exp( y );
}
```

Figure 5.19. Functions, prototypes, and calls.

This is a function call graph for the functions, prototypes, and calls of the program in Figure 5.19. Shading around a boxes indicates that the function is in a standard library.

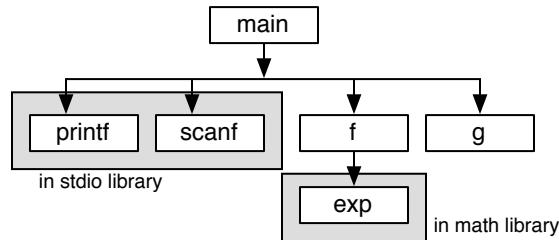


Figure 5.20. Function call graph with two levels of calls.

***First box: things that precede main() in a code module.***

- When the C compiler reaches the `#include` command, it puts a copy of the `tools.h` file into this program. This file contains prototypes for the functions in the `tools` library, including `banner()` and `bye()`, called in this program.
- Included files often contain other `#include` commands. For example, the `tools.h` file contains `#include` commands for the library header files, `stdio.h`, `math.h`, `string.h`, `time.h`, and `ctype.h`. If we include `tools.h` in a program, we need not write separate include commands for these other library header files.
- This program uses two constants, representing the minimum and maximum values acceptable for input. These constants are defined after the `#include` command and before the prototypes.
- We need a prototype for a function if a call on it comes before its definition in the file. Function `f()` is called (second box) from `main()` and defined after `main()` (fourth box). Therefore, `f()` needs a prototype, which is given on the fourth line of this box.
- The actual definition of function `g()` is given here, rather than a prototype. When a function definition comes before any use of that function, no prototype is needed. This often is done when a function is so simple that all its work is done in the `return` statement and so short that it can be written on one line.

***Second and third boxes: Calls on the programmer-defined functions.***

- We create `f()` and `g()` as two functions separate from `main()`, so that it is easy to change them when we need to do some other calculation. A good modular design keeps the calculation portion of a program separate from the user-interaction portion.
- In the second box, we set `z` equal to the result of calling function `f()` with the value of the variable `x`. Function `f()` was only prototyped before `main()`, so when the compiler reaches this box, the full definition of `f()` will not be known to it. However, the prototype for `f()` already was supplied, so the compiler knows that a call on `f()` should have one `double` argument and return a `double` result. This information is necessary to translate the call properly.
- In the third box, the function `g()` is called, and its return value is then passed directly to `printf()`. Since `g()` already was fully defined, the compiler has full knowledge of `g()` when it reaches this line and, therefore, is able to compile this call correctly.
- A sample output from this program, excluding the banner and closing comment, is

```
Enter a value of x between 10.50 and 87.00: 13.2
```

```
The value of x * exp(x) is: 7.13281e+06
```

```
The value of x * x + x is: 187.44
```

***Fourth box: Definition of programmer's function f().***

- Here we define `f()`. The return type and parameter list in the function definition must agree with the prototype given earlier.
- Function definitions should start with a blank line and a comment describing the action or purpose of the function. Discipline yourself to do this. The dashed line provides a visual separation and helps the programmer find the beginning of the function definition. This is extremely useful in a long program; make it a habit in your work.
- Compare this definition to the previous one-line definition of `g()`. The definition of `f()` begins with a descriptive header. The code itself is spread out vertically, with only one program element on each line, according to the accepted guidelines for good style. The definition of `g()` is written compactly on one line; that style is used only for very simple functions.

***Inner box: A call on a library function.***

- We call the function `exp()`, which is in the `math` library. We can do this because the header file, `math.h`, was included by `tools.h`, which was included in this file.
- The variable `y` is a `double`. The prototype for `exp()` says that its parameter is a `double`. The type mismatch here is not a problem. The compiler will note the mismatch and automatically compile code to convert the `double` value to a `double` format during the calling process.

This illustrates the function calls in Figure 5.19.

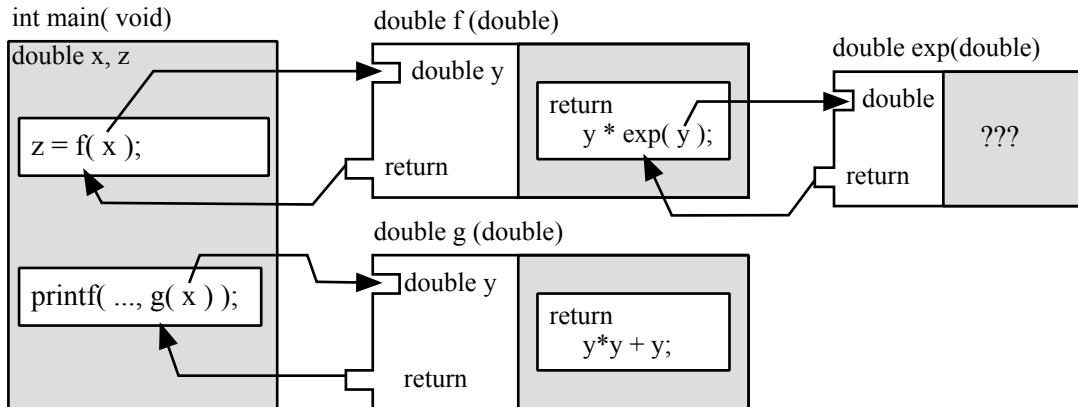


Figure 5.21. A function is a black box.

## 5.5 Application: Numerical Integration by Summing Rectangles

We introduce the topic of integration with a simple integration program where the function to be integrated is coded as part of the main program. The definite integral of a function can be interpreted as the area under the graph of that function over the interval of integration on the  $x$ -axis. If a function is continuous, we can approximate its integral by covering the area under the curve with a series of boxes and adding up the areas of these boxes. Several methods for numerical integration are based on a version of this idea:

- Divide the interval of integration into a series of subintervals.
- For each subinterval, approximate the area under the curve in that interval by a shape such as a rectangle or trapezoid whose area is easy to calculate.
- Calculate and add up the areas of all these shapes.

The simplest way to approximate the integral of a function is to use rectangles to approximate the area under the curve in each subinterval; the diagram in Figure 5.22 and the program in Figure 5.23 illustrate this approach. For each subinterval, we place a rectangle between the curve and the  $x$ -axis such that the upper-left corner of the rectangle touches the curve and the bottom of the rectangle lies on the  $x$ -axis. In this example, we calculate the integral of the function  $f(x)$ :

$$f(x) = x^2 + x, \quad \text{where } 0 \leq x \leq b$$

by summing 100 rectangular areas, each of width  $h = b/100$ , as shown by Figure 5.22. For example, the area of the rectangle between  $2h$  and  $3h$  is  $h \times f(2h)$ . Generalizing this formula, we get

$$\text{Area}_k = h \times f(k \times h)$$

Now, summing over 100 rectangles, we get

$$\text{Area} = \sum_{k=0}^{99} h \times f(k \times h)$$

### Notes on Figures 5.22 and 5.23: Integration by Rectangles.

**First box: the function definition.** This is the entire definition of the function we will integrate. Short functions can be written on one line.

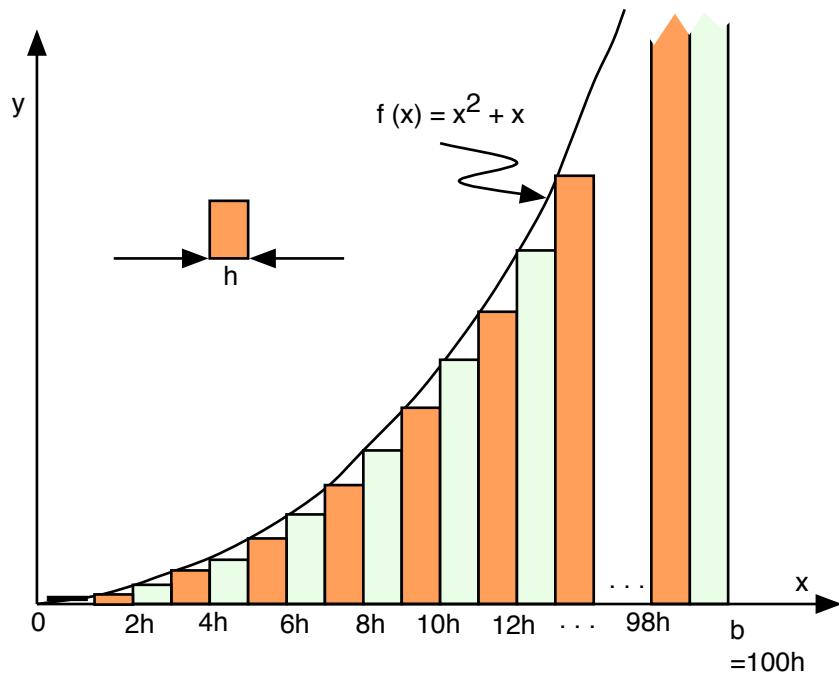


Figure 5.22. The area under a curve.

**Second box: declarations with initializations.**

- In this initial example, we integrate a single fixed function over a fixed interval using a fixed number of rectangles. Later, we present other integration programs that do the job in more general and more accurate ways.
- We integrate the function over the interval  $0 \dots 1$ .
- We divide this interval into 100 parts of length  $h = .01$ .

**Third box: initializations for the loop.**

- Although we could combine these initializations with the declarations, that is bad style. For trouble-free development of complex programs, all loop initializations should be placed immediately before the loop.
- To prepare for the loop, we initialize the `sum` that will accumulate the areas.
- The loop variable for this `while` loop is `k`. The first rectangle lies between  $a$  and  $a + 1 \times h$ , so we initialize `k` to 0. Since  $a = 0.0$ , the first rectangle we sum starts at  $x = 0.0 + 0 \times h = 0.0$ .

**Fourth box: the loop.**

- We execute the loop from `k = 0` until (but not when) `k = 100`. Therefore, we do it 100 times.
- The last rectangle summed starts at  $x = a + 99h$  and goes to  $x = a + 100h$ .
- We use the `++` operator to increment `k` each time around the loop.
- Note how convenient the `+=` operator is for adding a new term to a sum.

**Fifth box: the output.**

- The exact area under this curve for the interval  $0 \dots 1$  is  $.833333$ . Since we are calculating an approximate answer, it will be slightly different.
- The actual output is

```

#include <stdio.h>
#define N 100
double f( double x ){ return x*x + x; }

int main( void )
{
    double a = 0.0;           // Lower limit of integration.
    double b = 1.0;           // Upper limit of integration.
    double h = (b - a)/N; // Width of one of the N rectangles summed.

    double x;                 // Current function argument; a <= x < b.
    double sum;                // Total area of rectangles.
    int k;                     // Loop counter.

    printf( " Integrate x*x + x from %g to %g. \n", a, b );
    sum = 0.0;
    k = 0;

    while(k < N) {           // Add up N rectangles.
        x = a + k * h;       // Lower left corner of kth rectangle.
        sum += h * f( x );   // Area of rectangle at x.
        ++k;
    }

    printf( " The area is %g\n", sum );
    return 0;
}

```

Figure 5.23. Integration by summing rectangles.

```

Integrate x*x + x from 0 to 1.
The area is 0.82335

```

## 5.6 Functions with More Than One Parameter

### 5.6.1 Function Syntax: Two Parameters

We have studied a variety of `void` and non-`void` functions having either no parameters or one parameter and discussed the essentials of prototypes, parameters, and calls. Most functions, though, have more than one parameter, so we need to study the few remaining facets of the syntax for defining and calling functions with a more complex interface. The next few paragraphs summarize the syntax for functions with two parameters and a return value. The forms for three or more parameters follow the same pattern, with additional clauses added to the parameter list. We will use the following terminology in this discussion:

- **f-name** means any function name;
- **p-type** means the type of a parameter, **vp-name** means the name of a parameter,
- **r-type** means the type returned by the function,
- **var** means the name of a variable,
- **exp** means any expression of the correct type, possibly a simple variable name.

**Prototypes.** The fundamental form for the 2-argument prototype is:

```
r-type f-name( p1-type p1-name, p2-type p2-name );
```

However, parameter names are optional in a prototype, so we could also write the prototype like this:

```
r-type f-name( p1-type, p2-type );
```

For example, suppose we have a function named `cyl_vol` that takes two double parameters. Its prototype could be written in one of these two ways

```
double cyl_vol( double d, double h );
double cyl_vol( double, double );
```

In mathematical notation, this is called a  $\text{double} \times \text{double} \rightarrow \text{double}$  function.

**Call syntax.** A function call imitates the simpler form of the prototype; argument expressions of the appropriate type must be written, but parameter names are always omitted. The general syntax and a sample call on `cyl_vol` are:

```
General syntax: var = f-name( exp1, exp2 );
Example: volume = cyl_vol( diameter, height );
```

**Definition syntax.** A function definition imitates the longerform of the prototype; argument types and names must both be given. This is the general syntax and function header of `cyl_vol`:

```
General syntax: r-type f-name( p1-type p1-name, p2-type p2-name ){...
Example: double cyl_vol( double d, double h ){...
```

We illustrate these rules for the format of a two-parameter function using the program in Figures 5.24 and 5.25

#### Notes on Figures 5.24 and 5.25: Two parameters and a return value.

**First box: the prototypes.** We will use two  $\text{double} \times \text{double} \rightarrow \text{double}$  functions to calculate two properties of a cylinder: volume and surface area. The prototype for the first, `cyl_vol()`, is written with the optional parameter names and the prototype for the second function, `cyl_surf()` is written without. As always, either prototype could be written either way.

#### Second box: the first function call and output.

- Since `cyl_vol()` returns a value, calls will be in the context of an assignment statement or in the argument list of another function call. This call is part of an assignment.
- First, we call the function and save the answer in `volume`. Then, on the next line, we send the value of `volume` to `printf()`. The calculation could be combined with the output and condensed into one statement by putting the call on `cyl_vol()` directly into the argument list for `printf()`, thus:

```
printf( "\t The volume of this cylinder = %.2f \n",
        cyl_vol( diam, height ) );
```

- It is a matter of personal style which way you write this code. The one-line version is more concise and takes slightly less time and space. The two-line version, however, is easier to modify, works better with an on-line debugger, and enhances seeing and understanding the technical calculation.

**Third box: the second function call and output.** The calculation and output can be combined and condensed into one statement by putting the call directly into the argument list for `printf()`, as shown here.

#### Figure 5.25: definitions of cyl\_vol and cyl\_surf.

- Following modern guidelines for style, we prefer to write the return type, alone, on the first line of the function definition. The second line starts with the function name on the left. Using this style makes it somewhat easier to find the function names when you scan a long program and allows writing a comment about the return value. Of course, these two lines also could be combined, thus:

```
double cyl_vol( double d, double h ) { ...
```

---

This program illustrates the syntax for calling two-parameter non-void functions. The two functions called here are defined in Figure 5.25.

```

// -----
// Calculate the volume of a cylinder.
*/
#include <stdio.h>
#include <math.h>
#define PI 3.1415927

double cyl_vol( double d, double h );           // long form of prototype
double cyl_surf( double, double );               // short form of prototype

int main( void )
{
    double diam, height;                      // Inputs: dimensions of the cylinder.
    double volume;                            // Output: its volume.

    printf( "\n Calculate the Volume of a Cylinder\n\n"
            " Enter its diameter and height: " );
    scanf( "%lg%lg", &diam, &height );

    volume = cyl_vol( diam, height );
    printf( "\t The volume of this cylinder = %g\n", volume );

    printf( "\t Its surface area = %g \tn", cyl_surf( diam, height ) );
    return 0;
}

```

---

**Figure 5.24. Using functions with two parameters and return values.**

- The code in the body of this function uses only three variable names: `d`, `h`, and `r`. The first two are parameters to the function, the third is defined at the top of the function's block of code. All three objects are local to the function; that is, these variables are defined by the function and only this function can access them. Every properly designed function follows this pattern and confines its code to use only locally defined names. All interaction between the function and outside variables happens through the parameter list or the function's return value.

**Inner box: raising a number to a power.** Very few functions in the `math` library require two arguments; the most commonly used is the function `pow()`, which is used to raise a number to a power. In this box, we are calculating the square of `r`, the radius.

This function's prototype is `double pow( double, double)`. The first argument is the number to be raised, the second argument is the power to which to raise it. Both may be any `double` number. For example, `pow( 100, .5 )` raises 100 to the power .5, which is the same as calculating its square root.

#### **Output.**

```

Calculate the Volume of a Cylinder

Enter its diameter and height: 2.0 10.0
The volume of this cylinder = 31.4159
Its surface area = 69.115

Cylinder has exited with status 0.

```

We illustrate the syntax for definition of two-parameter non-void functions. These functions are called from Figure 5.24.

```
// -----
double                      // Calculate the volume of a cylinder
cyl_vol( double d, double h ) { // with diameter=d and height=h;
    double r = d / 2;          // r is the radius of the cylinder.
    return PI * pow( r, 2 ) * h;
}

// -----
double                      // Calculate surface area of cylinder
cyl_surf( double d, double h ) { // with diameter d and height h;
    double area_end, area_side;
    double r= d / 2;           // r is the radius of the cylinder.
    area_end = PI * pow( r, 2 ) // each end is a circle.
    area_side = h * 2 * PI * r; // length of side = circle perimeter
    return area_side + 2 * area_end; // surface is the side + 2 ends
}
```

Figure 5.25. Functions with two parameters and return values.

## 5.7 Application: Generating “Random” Numbers

### 5.7.1 Pseudo-Random Numbers

Many computer applications (experiments, games, simulations) require the computer to make some sort of random choice. To serve this need, programs called *pseudo-random number generators* have been devised. These start with some arbitrary initial value (or values), called the **seed**, and apply an algorithm to generate another value that seems unrelated to the first. Then this first result will be used as the seed for the next value and so on, as long as the user wishes to keep generating values.

The numbers generated by these algorithms are called **pseudo-random numbers** because they are not really random but the output of an algorithm and an initial value. If the same algorithm is run again with the same starting point, the same series of “random” numbers will be produced. The goal, therefore, is to find an algorithm and a seed that will produce a long series of numbers with no detectable pattern and without duplicating the seed. Repeating a seed would cause the series to enter a cycle.

The C function **rand()** generates pseudo-random integers, which might be 2 or 4 bytes long, depending on the local definition of type **int**. (The actual range of values is 0 ... **RAND\_MAX**, which is commonly the same as **INT\_MAX**.) This function does not implement the best known algorithm but is good enough for many purposes. The function is found in the **standard** library; to use it you must `#include <stdlib.h>`. Before calling **rand()** the first time, you must call another function, **srand()** to supply an initial seed value. This seed could be any integer value, such as a literal constant or a number entered by the user. However, in general, the user should not be bothered with selecting a seed, and a constant seed is undesirable because it always will result in the same pseudo-random series. (A constant seed can be useful during the debugging process so that error conditions can be repeated.) What therefore is needed for most applications is a handy source of numbers that are constantly changing and nonrepetitive. One such source is attached to most computers: the real-time clock. Therefore, it is quite common to read the clock to get an initial random seed. This technique is illustrated in Figure 5.26.

### 5.7.2 How Good Is the Standard Random Number Generator?

The next program generates a large quantity of random integers and counts the occurrences of 0. According to probability theory, if we generate numbers in the range  $0 \dots n - 1$ , approximately  $1/n$  of the values should be counted. No single program run can confirm whether the generator is fair. However, repeated trials or larger sample sizes will give some feeling for the quality of the random number generator being used. If the results are close to the expected value most of the time, the generator is performing well; otherwise, its behavior is questionable.

**Notes on Figure 5.26. Generating random numbers.**

*First box: initializing the random number generator.*

1. C provides a function in the `time` library that permits a program to read the system's real-time clock (if there is one). The return value of `time()` is an integer encoding of the time that has type `time_t` (an integer of some system-dependent length defined in `time.h`).
2. The argument in the call to `time()` normally is the address of a variable where we want the time stored. The function `time()` stores the current time into the given address in the same way that `scanf()` stores an input value into a variable whose address is given to it.<sup>13</sup> However, this function also returns the same time value through the normal function return mechanism. Since we only need this information once, we use a special constant value, `NULL`, as the argument; this is legal and tells the function that we don't want a second copy of the information stored anywhere in memory.
3. Our purpose here is not to know the actual time. Rather, we use the clock as a convenient source of a seed for the random-number generator. A good seed is an unpredictable number that never is the same twice, and the time of day suits this purpose very well.
4. The type cast operator, `(unsigned)`, in front of the call on the `time()` function is used to convert the `time_t` value returned by the `time()` function into the `unsigned int` form expected by `srand()`. Using an explicit cast instead of the standard automatic coercion eliminates a compiler warning message on some systems.

*Second box: data input and validation.* We eliminate divisors less than 2 because they are meaningless. We also set an arbitrary upper limit on the range of numbers that will be generated. Since the limit is relatively small, we can use a short integer to store it and there is space for printing many columns of numbers. Here is an example of the error handling:

```
Please choose n between 2 and 100: 1
Number is out of range.

Error exit; press '.' and 'Enter' to continue
```

A validation loop could be used here. We take the simpler approach of using `fatal()` because little effort has been invested so far in running this program and little is wasted by restarting it after an error.

*Large outer box: generating and testing the numbers.* This loop calls `rand()` many times and collects some information about the results. With the constant definitions given, we will generate 500 integers in the range  $0 \dots n - 1$ , where  $n \leq 100$ . These numbers will be printed in 10 columns. Occurrences of 0 will be counted.

*First inner box: generating the numbers.*

1. The function `rand()` returns a number between 0 and `RAND_MAX`. According to the standard, this number may vary, but it is at least 32,767. We must scale this number to the desired range  $0 \dots n-1$ .
2. The modulus operator is exactly what we want for a scaling operation, since its result is between 0 and the modulus  $-1$ . We compute `num % select` and store the result back in `num`.

---

<sup>13</sup>How this actually is done, using call by address, is discussed in Chapter 11.

---

We generate a series of pseudo-random numbers and print them in neat columns. When finished, we also print the number of zeros generated and the number expected, based on probability theory.

```
#include "mytools.h"
#define HOW_MANY 500 // Generate HOW_MANY random numbers
#define NCOL 10      // Number of columns in which to print the output.
#define MAX 100      // Upper limit on size of random numbers generated

int main( void )
{
    long num;          // a randomly generated integer
    short select;     // input: upper limit on range of random numbers
    short n;           // # of random numbers generated
    int count;         // # of zeros generated

    banner();

    srand( (unsigned) time( NULL ) );// seed random number generator.

    printf( " Generate %i random numbers in the range 0..n-1. \n"
            " Please choose n between 2 and %i: ", HOW_MANY, MAX );
    scanf( "%hi", &select );
    if (select < 2 || select > MAX) fatal( " Number is out of range." );

    // Generate random numbers and test for zeros. -----
    count = 0;          // Count zeros generated.
    for (n = 0; n < HOW_MANY; ) { // Generate HOW_MANY random numbers.

        num = rand();          // Generate a random long integer.
        num %= select;        // Scale to range 0..select-1.

        ++n;                  // Count the trials and...
        printf( "%5li", num ); // ...print all numbers generated.
        if (n % NCOL == 0) puts( "" ); // End line every NCOL outputs.

        if (num == 0) ++count; // ..count the zeros.
    }

    if (count % NCOL != 0) printf( "\n" ); // End last line of output.
    printf( "\n %i zeros were generated.", count );
    printf( "\n %.1f are expected on average.\n", HOW_MANY/(double)select );
    return 0;
}
```

---

Figure 5.26. Generating random numbers.

***Second inner box: lines of output.***

1. The counter `n` keeps track of the total number of random numbers produced so far.
2. We want the output printed in columns, so we use a fixed field width in the conversion specifier: `%7li`. Ten columns, each seven characters wide, will fit conveniently onto the usual 80-column line.
3. We want to print a '`\n`' after every group of `NCOL` numbers but not after every number. To do this, we count the output items as they are produced and print a newline character every time the counter is an even multiple of `NCOL`; that is, `n % NCOL == 0`.

***Third inner box: counting the zeros.*** To assess the “fairness” of the random-number generator, we can count the number of times a particular result shows up. If numbers in the range  $0 \dots n - 1$  are being generated, then each individual number should occur `HOW_MANY / n` times. In this program, we expect each number in the possible range  $1 \dots \text{select}-1$  to occur approximately  $500/\text{select}$  times. The following are the first and last lines of output from three runs. Note that the number of zeros generated on two of three trials differs substantially from the number expected. This is an indication that the `rand()` function does not produce a very even distribution of numbers on our computer.

```
Generate 500 random numbers in the range 0..n-1.
Please choose n between 2 and 100: 25
    7   19   24   8   18   20   8   10   4   5
    5   18   14   19   5   11   24   17   11   5
...
    2   0   5   21   22   3   21   7   23   18

    18   zeros were generated.
    20.0 are expected on average.
-----
Please choose n between 2 and 100: 33
...
    12   zeros were generated.
    15.2 are expected on average.
-----
Please choose n between 2 and 100: 33
...
    20   zeros were generated.
    15.2 are expected on average.
```

## 5.8 Application: A Guessing Game

In a classic game, one player thinks of a number and a second player is given a limited number of tries to guess it. The first player must say whether the guess is too small, correct, or too large. We illustrated a simple example of this game in Figure 6.31. Now, we implement the full game in the next program example, with the computer taking the part of the first player.

### 5.8.1 Strategy

Even if a person has never seen this game, it does not take long to figure out an optimal strategy for the second player:

- Keep track of the smallest and largest remaining possible value.
- On each trial, guess the number midway between them.

The computer’s response to each guess will eliminate half the remaining values, allowing the human player to close relentlessly in on the hidden number.

Figure 5.27 illustrates a game in which the range is  $1 \dots 1,000$  and the hidden number is 458. The player makes an optimal sequence of guesses, halving the range of remaining values each time: 500, 250, 375, 437, 468, 453, 461, 457, 459, 458. In this example, 10 guesses are required to home in on the hidden number. In fact, with this strategy, this also is the maximum number of trials required to find any number in the given range. Half the time the player will be lucky and it will take fewer guesses. The code for the program that implements this game is given in Figures 5.28 (`main()`) and 5.29 (`one_game()`, which handles the sequence of guesses).

In this example, the total range of possible values is  $1 \dots 1,000$ . The hidden number,  $num = 458$ , is represented by a dashed line. The solid vertical lines represent the guesses of a player using an optimal strategy; only the first five guesses are shown.

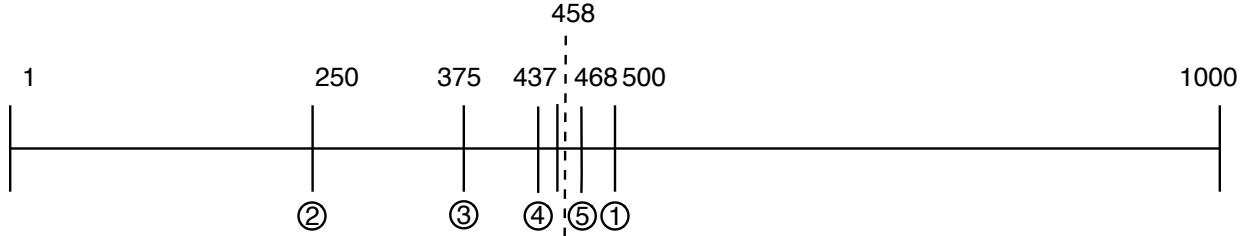


Figure 5.27. Halving the range.

### 5.8.2 Playing the Game

**Choosing and scaling the number.** We can call `rand()`, as was done in Figure 5.26, to get a number in the range  $0 \dots \text{INT\_MAX}$ . In this game, however, we require a random number between 1 and 1,000, so the number returned by `rand()` must be scaled and adjusted to fall within the desired range. To do this, we use the `%` (modulus) operator. For instance, `rand() % TOP` gives us a random number in the range  $(0 \dots \text{TOP}-1)$ . We then adjust it to the desired range simply by adding 1. This formula is used in the first box of Figure 5.29.<sup>14</sup>

**Setting a limit on the number of guesses.** The strategy shown in Figure 5.27 is an example of an important algorithm called **binary search**,<sup>15</sup> because we search for a target value by dividing the remaining set of values in half. If  $N$  values were possible in the beginning, the second player always could discover the number in  $T$  trials or fewer, where  $N \leq 2^T$ . Another way to say this is that

$$T = \lceil \log_2 N \rceil$$

where  $\lceil \dots \rceil$  means that we round to the next higher integer. Paraphrased, the maximum number of trials required will be the base-2 logarithm of the number of possibilities, rounded up to the next larger integer. In our case, this is

$$\lceil \log_2 1000 \rceil = \lceil 9.96578 \rceil = 10$$

To introduce an element of luck into the game, set the maximum number of guesses to something smaller than  $T$ . This, in fact, is what we do in Figure 5.28; we round *down* instead of *up*, allowing too few guesses about half of the time. This “stacks” the game in favor of the computer.

**Calculating a base-2 logarithm.** The C math library provides two logarithm functions: one calculates the base-10 log, the other the natural log (base  $e$ ). Neither of these is what we want, but you can use the natural log function to compute the log of any other base,  $B$ , by the following formula:

$$\log_B(x) = \frac{\log_e(x)}{\log_e(B)}$$

In C, the natural log function is named `log()`, so to calculate the base-2 log of 1,000, we write

$$\log(1000) / \log(2)$$

This formula is used in the second box of Figure 5.28.

#### Notes on Figure 5.28. Can you guess my number?

<sup>14</sup>For reasons too complex to explain here, this formula has a slight bias toward lower numbers in the range. However, if the range is small compared to `RAND_MAX`, the bias is insignificant.

<sup>15</sup>Other examples of binary search are given in Chapters 11 and 19.

---

This main program calls the function in Figure 5.29. It repeats the game as many times as the player wishes.

```
#include "mytools.h"
void one_game( int tries );

#define TOP 1000           // Top of guessing range.

int main( void )
{
    int do_it_again;          // repeat-or-stop switch

    const int tries = log( TOP ) / log( 2 );      // One too few.

    banner();
    puts( "\n This is a guessing game.\n I will "
          "think of a number and you must guess it.\n" );

    srand( (unsigned)time( NULL ) );      // seed number generator.

    do { one_game( tries );
        printf( "\n\n Enter 1 to continue, 0 to quit: " );
        scanf( "%i", &do_it_again );
    } while (do_it_again != 0);

    return 0;
}
```

---

**Figure 5.28. Can you guess my number?**

***First box: guessing range.***

In this game, the player will try to guess a number between 1 and TOP.

***Second box: the number of trials.***

- We calculate `tries`, the maximum number of trials the user is allowed. It is defined as a **constant** because it depends only on `TOP` and does not change from one game to another. We use a `const` variable, rather than `#define`, because the definition is not just a simple number.<sup>16</sup>
- The result of the division operation is a *double* value that is coerced to type `int` when stored in `tries`.
- C permits us to use a formula to define the value of a constant. Such formulas can use literal constants (such as 2) and globally defined symbols (such as `TOP`). Inside a function definition, the parameter values also can be used in a constant expression.
- We calculate the base-2 logarithm of the number of possible hidden values and use this to set the maximum number of guesses. As described, we set this maximum so that the player will succeed about half the time. The rest of the time, the player will be one guess short of success, even if he or she is using the optimal search strategy.

***Third box: initializing the random number generator.*** As in Figure 5.26, we initialize C's random number generator with the current time of day.

**Notes on Figure 5.29. Guessing a number.** The `one_game()` function is called from `main()` in Figure 5.28 for each round of the game that the player wishes to play.

---

<sup>16</sup>For reasons beyond the scope of this book, this is more efficient.

---

This function is called from Figure 5.28. It plays the number-guessing game once.

```

void
one_game( int tries )
{
    int k;                                // Loop counter.
    int guess;                             // User's input.

    const int num = 1 + rand() % TOP;        // The hidden value.

    printf( " My number is between 1 and %i;"      "
           " I will let you guess %i times.\n"
           " Please enter a guess at each prompt.\n", TOP, tries );

    for (k = 1; k <= tries; ++k) {
        printf( "\n Try %i: ", k );
        scanf( "%i", &guess );
        if (guess == num) break;
        if (guess > num) printf( " No, that is too high.\n" );
        else printf( " No, that is too low.\n" );
    }

    if (guess == num) printf( " YES!! That is just right. You win! \n" );
    else printf( " Too bad --- I win again!\n" );
}

```

---

**Figure 5.29. Guessing a number.**

**First box: the hidden number.** The first thing this function does is choose a hidden value. The value is defined as a constant because it does not change from the beginning of a round to the end of that round. In this case, the constant expression involves the value of a global constant. To calculate a random hidden number, we use the random number generator `rand()`, scaling and adjusting its value to the required range, as discussed earlier in this section.

**Second outer box: playing the game.** The code in this box is almost identical to the earlier version in Figure 6.31. Correct guesses are handled by an `if...break` in the inner box. When the loop exits, the program prints a failure message if the most recent guess was wrong.

Here is some sample output (omitting output of the query loop). The first two lines were printed by the main program, the rest of the dialog was printed by the `one_game()` function. In this sample game, the player was lucky and guessed the hidden number in only five tries. We expect this to happen only once in every 32 games.

This is a guessing game. I will think of a number and you must guess it.

My number is between 1 and 1000; I will let you guess 9 times.  
Please enter a guess at each prompt.

Try 1: 500  
No, that is too high.

Try 2: 250  
No, that is too high.

Try 3: 125

No, that is too high.

Try 4: 62

No, that is too low.

Try 5: 93

YES!! That is just right. You win!

## 5.9 What You Should Remember

### 5.9.1 Major Concepts

- Functions can be used to break up programs into modules of manageable complexity. In this way, a highly complex job can be broken into short units that interact with each other in controlled ways. These modules should have the following properties:
  - The purpose of each function should be clear and simple.
  - All its actions should hang together and work at the same level of detail.
  - A function should be short enough to comprehend in its entirety.
  - The length and complexity of a function can be minimized by calling other functions to do subtasks.
- Function definitions can come from the standard C libraries such as the `stdio` and `math` libraries or from a personal library. Functions also can be defined by the programmer.
- Some functions compute and return values; others do not. A function with no return value is called a `void` function. Such functions normally perform input or output of some sort. This chapter presented programmer-defined functions with zero, one, and two parameters (`void`→`void`, `double`→`double`, and `double`×`double`→`double` respectively). Additional types of library functions and programmer-defined functions are introduced in subsequent chapters as additional data types are discussed.
- A call to a function that returns a value normally is found in an assignment statement, an expression, or an output statement.
- The basic components of a function are the prototype and the function definition, which consists of a header and a body.
- Arguments are the means by which a calling program communicates data to a function. Parameters are the means by which a function receives the communicated data. When a function is called, the actual arguments in the function call are passed into the function and become the values of the function's formal parameters.
- After passing the parameter values, control is passed into the function. Computation starts at the top and proceeds through the function until it is finished, at which time the result is returned and control is transferred back to the calling program. A more detailed look at functions will be given in Chapter 9.
- The name of the parameter in a function does *not* need to be the same as the name of a variable used in calling that function.
- Functions allow an application program to be constructed and verified much more easily than a similar program written as one massive unit. Each function, then each module, is developed and debugged before it is inserted into the final, large program. This is how professional programmers have been able to develop the large, sophisticated systems that we use today.
- Here is a (questionable) rhyme to help you remember the four ways to extract an integer from a `double` value:

`ceil()` goes up and `floor()` goes down; `rint()` goes nearby but assign doesn't round.

- Random numbers. Applications such as games, experiments, and quiz programs require a program to make a series of randomized selections from a preset list of numbered options. To do this, we use an algorithm called a *pseudo-random number generator*, which generates a series of integers with no apparent pattern. The random number then is scaled to be in the proper range if it does not fall within the range of selection numbers.

The standard library provides the functions `srand()` and `rand()`, which together implement a pseudo-random number generator.

### 5.9.2 Local Libraries and Header Files

At various places in this chapter, suggestions were made about building a personal library called “mytools”. In this section, we gather together the various parts that could be included in that library and show how they would be organized into a header file `mytools.h` and a code file `mytools.c`, and how a client program would use such a library.

**The header file for mytools.**

```
// -----
// Alice Fischer's personal tools library.           File: mytools.h
// Last updated on July 29, 2011.
// Standard library headers that I need. -----
#include <stdio.h>      // for puts(), printf(), and scanf()
#include <stdlib.h>      // for exit()
#include <time.h>        // for time_t, time() and ctime()
#include <stdbool.h>     // for bool data type; c99 standard

// defined on some systems but not on mine.
#define PI 3.14159265358979323846264338327951

// Prototypes for my own tool functions. -----
void banner( void );           // Print a neat header for the output.
```

**The code file for mytools.**

```
// -----
// Alice Fischer's personal tools library.           File: mytools.c
// Last updated on Tue Sep 16 2003.

#include "mytools.h"
// -----
void banner( void )           // Print a neat header for the output.
{
    time_t now = time(NULL);
    printf( "\n-----\n" );
    printf( "    Alice E. Fischer\n    CS 110\n    " );
    printf( ctime( &now ) );
    printf( "-----\n" );
}
```

**A program that uses mytools.**

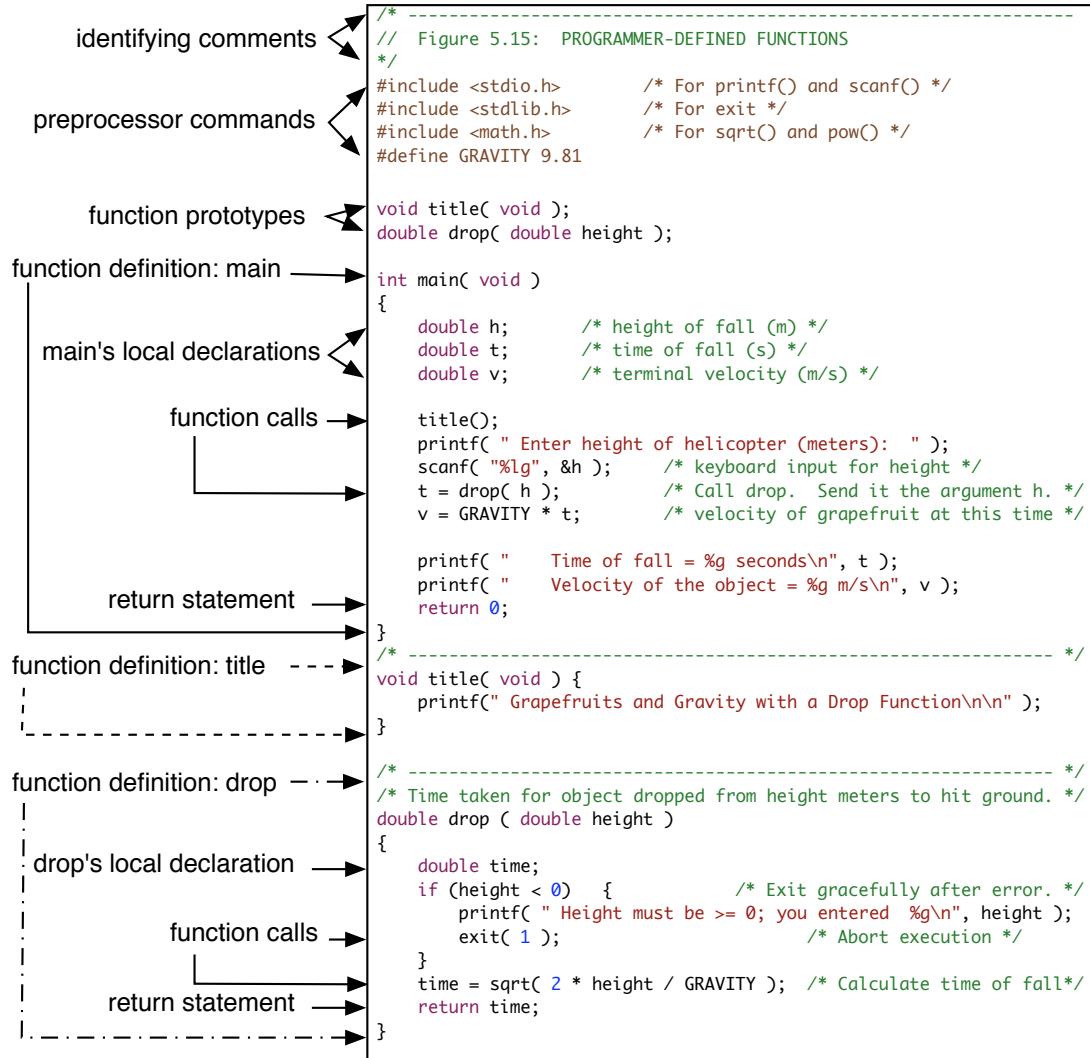
```
// -----
// Demo program for using a personal library. Both files mytools.h and
// mytools.c must be added to the project for this program.
// -----
#include "mytools.h"
int main( void )
{
    int count; // Number of people in the room.
    banner();
    printf( " Demonstrating the use of a personal library.\n"
            " How many people are watching? " );
    scanf( "%i", &count );
    if (count < 2) puts( " Not enough.\n" );
    else puts( " Hello you-all!\n" );
    return EXIT_SUCCESS;
}
```

Sample output from this demo.

```
-----
Alice E. Fischer
CS 110
Tue Sep 16 12:26:45 2003
-----
Demonstrating the use of a personal library.
How many people are watching? 3
Hello you-all!
```

### 5.9.3 The Order of the Parts of a Program

The following diagram summarizes the order in which it is customary to write prototypes, function definitions, and other elements in a source code file. The prototypes are normally written first, followed by the `main()` function then all other functions, in any convenient order.



#### 5.9.4 Programming Style

- Keep the main program and all function definitions short. An entire function should fit on one video screen, so that the programmer can see the declarations and the code at the same time. When a function begins to get long, this often is a sign that it should be broken into two or more functions.
- Every function definition should start with a distinctive and highly visible comment. We use a dashed line followed by a brief description of the purpose of the function. This comment block helps you to find each portion of your program quickly and easily, both on screen and on paper.
- Function names should be descriptive, distinctive, and not overly long.
- All the object names used in a function should be either parameter names or defined locally as variables. This provides maximum isolation of each function from all others, which substantially aids debugging.

#### 5.9.5 Sticky Points and Common Errors

- The prototype must match the function header. If there is a mismatch, the program will not compile correctly.
- Prototypes end in semicolons. If the semicolon is missing, the compiler “thinks” that the prototype is the function header. This will cause many meaningless error messages.

- Definitions must *not* have a semicolon after the function header. If a semicolon is written there, the compiler “thinks” the header is a prototype and will give an error comment on the next line.
- If the prototype is missing or comes after the first function call, the compiler will construct a prototype using the types of the parameters given in the first function call. The return type always will be `int`, which may or may not be correct. If the constructed prototype is wrong, it will cause the compiler to give error comments on lines that are correct.

### 5.9.6 Where to Find More Information

- Chapter 9 presents the full process of top-down programming and stub testing. This discussion has been deferred until the basic mechanics of functions and function calls are better understood.
- Chapter 9 has a more complete discussion of functions and prototypes.
- Chapter 9 explains local and non-local variables and how they are implemented using activation records on the run-time stack. Figure 9.2 shows how storage might be managed at run time for the cylinder program in Figure 5.24.
- Functions and type definitions to help programmers use the clock are described in Appendix F and parts are discussed in detail in Chapter 12.
- Chapter 12 shows a revised version of the `banner()` function.
- The website for this chapter introduces a function named `fatal()` that combines the actions of `printf()` and `exit()`, allowing error handling to be done in one statement.
- The results of a subprogram must be passed back to the caller. Depending on the function’s purpose, there may be no, one, or more results, but only one result can be returned through a `return` statement. There are several ways around this difficulty:
  - Chapter 10 discusses array parameters, which can be sent, empty, into a function, then filled in the function and returned containing a potentially large amount of data.
  - Chapter 11 shows how pointer arguments may be used to return information from a function to the caller.
  - Chapter 13 discusses compound objects (structures) that can contain and return many pieces of information packed into a single object.

### 5.9.7 New and Revisited Vocabulary

These are the most important terms and concepts that were introduced or discussed in this chapter:

function	function comment block	call graph
standard library	function body	pseudo-random numbers
personal library	function return	seed
header file	programmer-defined function	binary search
function call	prototype declaration	constant expression
caller	function definition	void→int function
argument	function interface	void→void function
subprogram	function header	int→void function
transfer of control	parameter	double→double function
interrupted flow	local variable	double×double→double function
exception		

The following C keywords, header files, library functions, constants, and types were discussed in this chapter:

return statement	<stdlib.h>	time_t (from time.h)
<stdio.h>	EXIT_FAILURE (from stdlib.h)	time(NULL) (from time.h)
scanf() (from stdio)	EXIT_SUCCESS (from stdlib.h)	ctime() (from time.h)
printf() (from stdio)	exit() (from stdlib)	<math.h>
puts() (from stdio)	abs() (from stdlib)	sqrt() (from math)
<limits.h>	srand() (from stdlib)	floor() (from math)
INT_MAX (from limits.h)	rand() (from stdlib)	log() (from math)
RAND_MAX (from stdlib.h)	<time.h>	pow() (from math)

## 5.10 Exercises

### 5.10.1 Self-Test Exercises

- For each part, write a double→double function that computes the formula and returns the result:
  - Tangent of angle  $x = \frac{\sin(x)}{\cos(x)}$
  - Surface area of a sphere with radius  $r = 4 \times \pi \times r^2$
- For each part, write a double×double→double function that computes the formula and returns the result:
  - Hypotenuse of a right triangle: length =  $\sqrt{\text{base}^2 + \text{height}^2}$
  - Polar to rectangular coordinates:  $y = r \times \cos(\theta)$
  - Rectangular to polar coordinates:  $\theta = \text{arc tangent}(y/x)$  for  $x \neq 0$
- Write the prototype that corresponds to each function definition:
  - double cube( double x ) { return x\*x\*x; }
  - void three\_beeps() { beep(); beep(); beep(); }
  - int surprise( void ) { return 17; }
  - int ratio( double a, double b ) { return a/b; }
- Explain the difference between
  - An argument and a parameter
  - A prototype and a function header
  - A header file and a source code file
  - A personal library and a standard library
  - A function declaration and a function call
  - A function declaration and a function definition
- What happens on your compiler when a prototype for a programmer-defined function is omitted? To find out, start with the code from Figures 5.15 and 5.18 and delete the prototype above the main program.
- Prototypes and calls. Given the prototypes and declarations that follow, fix the errors in each of the lettered function calls.

```

void    squawk( void );
int     half( int );
double  area( double, double );
int j, k;
double x, y, z;
  
```

- `k = squawk();`
- `squawk( 3 );`
- `j = half( 5, k );`

- (d) `j = half( int k );`
- (e) `y = area( double 3.0, x );`

7. Find the error here. Using the declarations given in problem 6, find the one function call below that has an error and explain what the error is. (The other three are correct.)

- (a) `y = area( x+2, 3 );`
- (b) `y = half( half( k ) );`
- (c) `printf( "%g %g", x, half( x ) );`
- (d) `y = area( sin( x ), z );`

### 5.10.2 Using Pencil and Paper

1. List everything you must write in your program when you want to *use* (not define) each of the following:
  - (a) A programmer-defined void→void function named `help()`
  - (b) The `fatal()` function
  - (c) The `sqrt()` function
2. Write a prototype that could correspond to each function call below:
  - (a) `do_it();`
  - (b) `x = calculate( y, z );`
  - (c) `k = get_Integer();`
  - (d) `x = cubeRoot( y );`
3. For each part, write a double→double function that computes the formula and returns the result:
  - (a) Sine:  $\sin(x) = \sqrt{1 - \cos^2(x)}$
  - (b) Tangent:  $\tan(2x) = \frac{2 \tan(x)}{1 - \tan^2(x)}$
  - (c) Diagonal of a square with side  $s = \sqrt{2 \times s^2}$
  - (d) Volume of a sphere with radius  $r = \frac{4\pi}{3} \times r^3$
4. Write a double×double→double function that computes the formula and returns the result:  
 $\text{Sum of angles: } \sin(x + y) = \sin(x) \cos(y) + \cos(x) \sin(y)$

5. What happens on your compiler when a required `#include` command is omitted? Will a program compile? Will it work correctly?
6. Prototypes and calls. Given the prototypes and declarations that follow, say whether each of the lettered function calls is legal and makes sense. If the call has an error, fix it.

```
void    squawk( void );
int     half( int );
double  area( double, double );
int j, k;
double x, y, z;
```

- (a) `j = squawk();`
- (b) `half( k );`
- (c) `j = half( 5 * k );`
- (d) `y = area( double 3.0, double x );`
- (e) `y = area( z );`
- (f) `y = area( pow( x, 2 ), z );`

(g) `scanf( "%i %i", &k, &half( k ) );`  
 (h) `y = area( x, y, z );`

7. Write the prototype that corresponds to each function definition. Then define appropriate variables and write a legal call on the function.

```
(a) double inches( double cm ) { return cm / 2.54; }
(b) void beeps( int n )
    {
        int k = 0;
        while (k < n) {
            beep();
            ++k;
        }
    }
```

### 5.10.3 Using the Computer

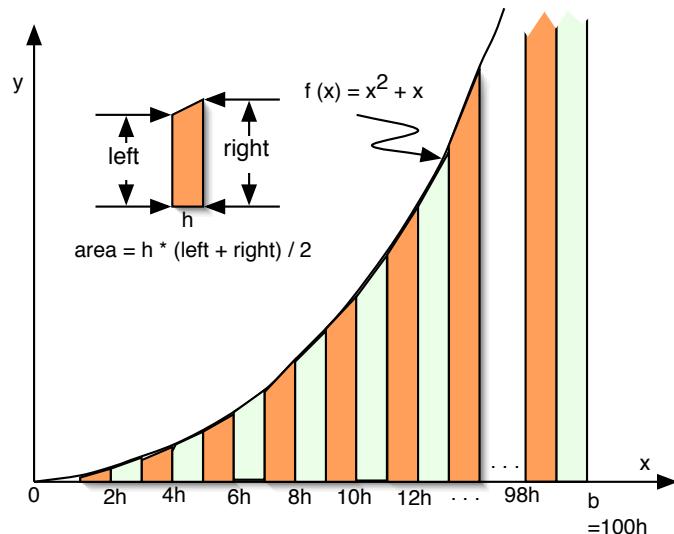
1. Geometric mean.

A *geometric progression* is a series such as 1, 2, 4, 8, ... or 1, 5, 25, 125, ... such that each number in the series is multiplied by a constant to get the next number. More formally, the constant,  $R$ , is called the **common ratio**. If the first term in the series is  $a$ , then succeeding terms will be  $a * R$ ,  $a * R^2$ ,  $a * R^3$ , ... For instance, if  $a = 1$  and  $R = 2$ , we get 1, 2, 4, 8, ... and if  $a = 1$  and  $R = 5$ , we get 1, 5, 25, 125, ...

Given terms  $k - 1$  and  $k + 1$  in a geometric progression, write a main program and at least one function to compute term  $k$  and the common ratio,  $R$ . Term  $k$  is called the *geometric mean* between the other two terms. The formulas for the  $k$ th term and common ratio are

$$R = \sqrt{\frac{t_{k+1}}{t_{k-1}}} \quad \text{and} \quad t_k = R \times t_{k-1}$$

2. Integration using trapezoids.



Modify the program in figure 5.23 to use trapezoids, rather than rectangles, to approximate the area under the curve, according to this sketch of the trapezoid method: Compare your answers to those from Figure 5.23. What can you say about their accuracy?

3. Numerical integration.

The program in Figure 5.23 integrates the function  $f(x) = x^2 + x$  for  $0 \leq x \leq 1.0$ . Modify this program to integrate the function  $f(x) = x^2 + 2x + 2$  for  $-1.0 \leq x \leq 1.0$ . If you have studied symbolic integration, compare your result to the exact analytical answer.

4. Precision of numerical computations.

Figure 5.23 gives a program that integrates a function by rectangles. Keep the function `f()` and modify the main program so that the integration process will be repeated using 10, 20, 40, 80, 160, and 320 rectangles. Print a neat table of the number of rectangles used and the results computed each time. Compare the answers. What can you say about their accuracy?

5. Sales tax.

Write a double→double function whose parameter is a purchase price. Calculate and return  $T$ , the total price, including sales tax. Define the sales tax rate as a `const R` whose value is 6%. Write a main program that will read  $P$ , the before-tax amount of a purchase, call your function to calculate the after-tax price, and print out the answer. Both prices will be in units of dollars and cents. What are the appropriate types for  $P$ ,  $R$ , and  $T$ ? Why? Use the program in Figure 5.15 as a guide.

6. Fence me in.

A farmer has several rectangular fields to fence. The fences will be made of three strands of barbed wire, with fence posts no more than 6 feet apart and a stronger post on every corner. Write a complete specification, with diagram, for a program that will input the length and width of a field, in feet. Make sure that each input is within a meaningful range. If so, calculate the area of the field and the total length of barbed wire required to go around the field. Also calculate the number of fence posts needed (use the `ceil()` function from the `math` library). Now write a program that will perform the calculations and display the results.

7. A spherical study.

Write four functions to compute the following properties of a sphere, given a diameter,  $d$ , which is greater than or equal to 0.0:

- (a) Radius  $r = d/2$
- (b) Surface area  $= 4 \times \pi \times r^2$
- (c) Circumference  $= \pi d$
- (d) Volume  $= \frac{4\pi}{3} \times r^3$

Write a main program that will input the diameter of a sphere, call all four functions, and print out the four results. Do not accept inputs less than 0.0.

8. Take-home pay.

Write a double→double function whose parameter is an employee's gross pay for one month. Compute and return the take-home pay, given the following constants:

- Medical plan deduction = \$75.65
- Social security tax rate = 7.51%
- Federal income tax rate = 16.5%
- State income tax rate = 4.5%
- United Fund deduction = \$15.00

The medical deduction must be subtracted from the gross pay before the tax amounts are computed. Then the taxes should be computed and subtracted from the gross. As each one is computed, print the amount. Finally, subtract the United Fund contribution and return the remaining amount. Your main program should print the final pay amount.

9. Hourly Employee.

Write a double×double→double function whose parameters are an employee's hourly pay rate and number of hours worked per week. Compute and return the gross pay.

Your main program should call the hourly-pay function, then use its result to call the take-home pay function described in the previous problem. Print the take-home pay, as before.

## 10. Compound interest.

- (a) Write a function with three double parameters to compute the amount of money,  $A$ , that you will have in  $n$  years if you invest  $P$  dollars now at annual interest rate  $i$ . The formula is

$$A = P(1 + i)^n$$

- (b) Write a main program that will permit the user to enter  $P$ ,  $i$ , and  $n$ . Call your function to compute  $A$ . Your main program should echo the inputs and print the final dollar amount.

## 11. Probability.

A statistician needs to evaluate the probability,  $p$ , of the value  $x$  occurring in a sample set with a known normal distribution. The mean of the distribution is  $\mu = 10.71$  and the standard deviation is  $\sigma = 1.14$ .

- (a) Write a double→double function with parameter  $x$  that computes the value of the probability formula for a normal distribution, which follows. To compute  $e^x$ , use the `exp()` function from the math library; its prototype is `double exp( double x )`.

$$p = \frac{1}{\sigma \times \sqrt{2\pi}} \times e^{-d}, \quad \text{where } d = \frac{[(x - \mu)/\sigma]^2}{2}$$

- (b) Write a main program to input the value for  $x$ , call your probability function, and print the results.



# Chapter 6

# More Repetition and Decisions

This chapter continues the discussion of control statements, which are used to alter the normal top-to-bottom execution of the statements in a program. As each new kind of statement is covered, its corresponding flow diagram will be shown. We present the syntax, flowcharts, and examples of use for these statements.

There are three kinds of loop statements in C, each of which is used to repeat a block of code. The `while` loop was introduced in Chapter 3. This chapter presents two additional loop statements, `for` and `do...while`. We also discuss various ways that loops can be used.

Conditional control statements are used to determine whether to execute or skip certain blocks of code. The C language has three kinds of conditional control statements: the `if...else` statement and the simple `if` statement without an `else` clause, which are introduced in Chapter 3, and a multibranched conditional statement, the `switch` statement. In this chapter we examine the `switch` statement and a new way to use the simple `if` statement.

We briefly cover the `break` and `continue` statements, which interrupt the normal flow of control by transferring control from inside a block of code to its end or its beginning. C also supports the `goto` control statement. However, we neither explain it nor illustrate its use, because it is almost never needed in C and its unrestricted nature makes it error prone. Using a `goto` statement is considered very poor programming practice because it leads to programs that are hard to both debug and maintain.

## 6.1 New Loops

### 6.1.1 The `for` Loop

The `for` loop in C implements the same control pattern as the `while` loop; anything that can be done by a `while` loop can be done in the same way using a `for` loop and vice versa. A `for` loop could be thought of as a shorter, integrated notation for a `while` loop that brings together the initialization(s), the loop test, and the update step in a single header at the top of the loop, as shown in Figure 6.1.

Thus, `for` is not really a necessary part of the language. However, it probably is more widely used than either of the other loops because it is very convenient and it captures the nature of a loop that is controlled by a counter. After gaining some experience, most programmers prefer using the `for` statement for many kinds of loops, especially for **counted loops** like that in Figure 6.1. An extremely simple program containing a `for` loop is shown, with its output, in Figure 6.2. This illustrates how a loop should be laid out in the context of a program.

**The syntax and flow diagram for `for`.** The `for` statement has a header consisting of the keyword `for`, followed by a parenthesized list of three expressions separated by semicolons: an initialization expression, a condition, and an update expression. All three expressions can be arbitrarily complicated. Usually, however, the first part is a single assignment, the second is a comparison, and the third is an increment expression, as shown in Figure 6.1.

Since `for` and `while` implement very similar control structures, a `for` loop can be diagrammed in the same manner as a `while` loop: using separate boxes for the initialization, test, and update steps. However, there is

In this simple `while` statement and its corresponding `for` statement, `k` is used as a counter to keep track of the number of loop repetitions. When `k` reaches 10, the body will have been executed 10 times and the loop will exit. Following the loop, the values of `k` and `sum` will be printed. The output will be 10 45.

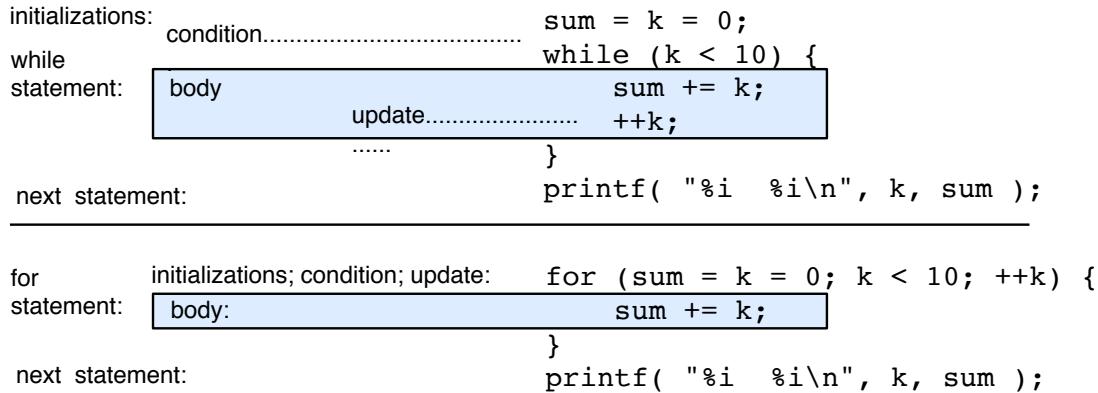


Figure 6.1. The `for` statement is a shorthand form of a `while` statement.

a more compact, single-box diagram that combines these pieces into one multipart control box with a section for each part of the loop header. (See Figure 6.3). Control enters through the initialization section at the top, then goes through the upper diagonal line into the condition section on the right. If the condition is true, control leaves through the lower exit, going through the boxes in the body of the loop, coming back into the update section of the `for` box, and finally, going through the lower diagonal line into the test again. If the test is false, control leaves the loop through the upper exit. The flow diagrams in Figure 6.4 compare the equivalent `while` and `for` loops of Figure 6.1. Both the statement syntax and the diagram for the `for` loop are more concise—fewer parts are needed and those parts are more tightly organized—than in a corresponding `while` loop.

**Declaring the for-loop variable.** If the loop variable will be used *only* within the loop, it is customary to declare it in the parentheses that follow the keyword `for`, as shown in Figure 6.2. This style keeps the declaration near the use of the variable and is consistent with modern usage in Java and C++.

**Initializing the loop variable.** An initialization statement must be written before a `while` loop so that the loop variable has some meaningful value before the loop test is performed the first time. In a `for` loop, this initialization is written as the first expression in the loop header. When the `for` loop is executed, this expression will be evaluated only once, before the loop begins.

---

```

// -----
// Counting with a for loop.
#include <stdio.h>                               The output is
int main( void )                                     0
{
    for (int k = 0; k < 5; ++k) {
        printf( " %i \n", k );
    }
    puts( "-----\n" );
}

```

---

Figure 6.2. A simple program with a `for` statement.

The general form of a **for** flow diagram is shown here. Control passes through the loop in the order indicated by the arrows. Note that this order is not the same as the order in which the parts of the loop are written in the code.

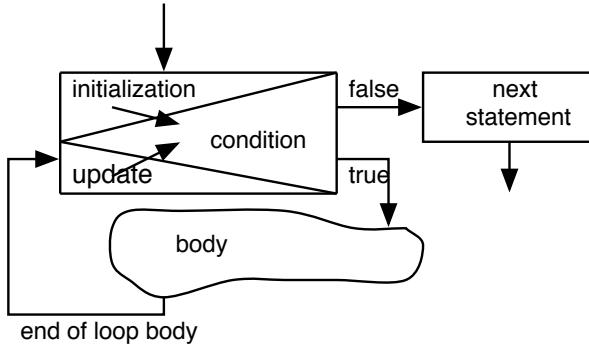


Figure 6.3. A flow diagram for the **for** statement.

The **,** (comma operator) can be used in a **for** loop to permit more than one item to be initialized or updated. For example, the loop in Figure 6.1 could be written with separate initializations for **sum** and **k**:

```
for (sum = 0, k = 0; k < 10; ++k) ...
```

Technically, both assignment and comma are operators. They build expressions, not complete statements, so we are permitted to use either or both any time the language syntax calls for an expression.

**The loop test and exit.** The condition in a **for** statement obeys the same rules and has the same semantics as that of a **while** condition. It is executed after the initializations, when the loop is first entered. On subsequent trips through the loop, it is executed after the update. If the condition does computations or causes side effects, such as input or output, those effects will happen each time around the loop. (This programming style is not recommended. It may be concise, but it sacrifices clarity and does not work well with some on-line debuggers.) Finally, note that the body of the loop will not be executed at all if the condition is false the first time it is tested.

The flow diagram for the **while** loop in Figure 6.1 is shown on the left and the diagram for the **for** loop is on the right. Note that the parts of the two loops are executed in exactly the same order, as control flows into, through, and out of the boxes along the paths indicated by the arrows.

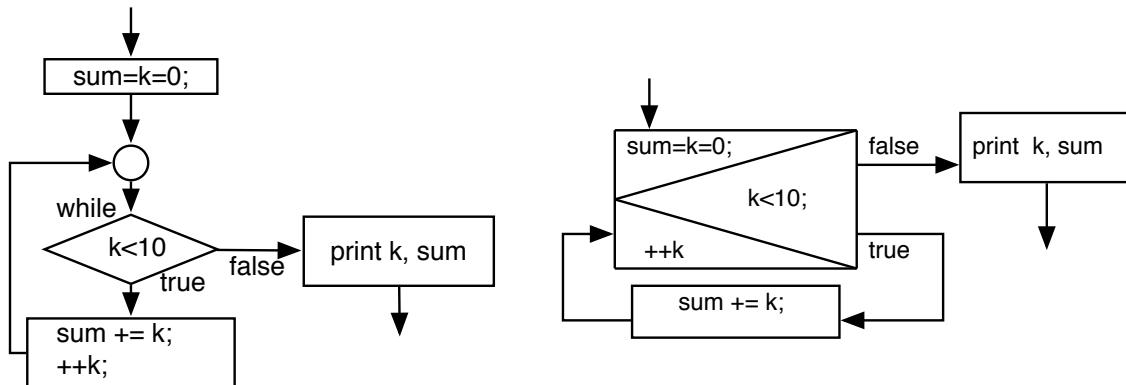


Figure 6.4. Diagrams of corresponding **while** and **for** loops.

The general form of a `do...while` flow diagram is shown on the left. Control passes through the loop body before reaching the test at the end of the loop. On the right is a diagram of a summing loop, equivalent to the loops in Figure 6.4 that use `while` and `for` statements.

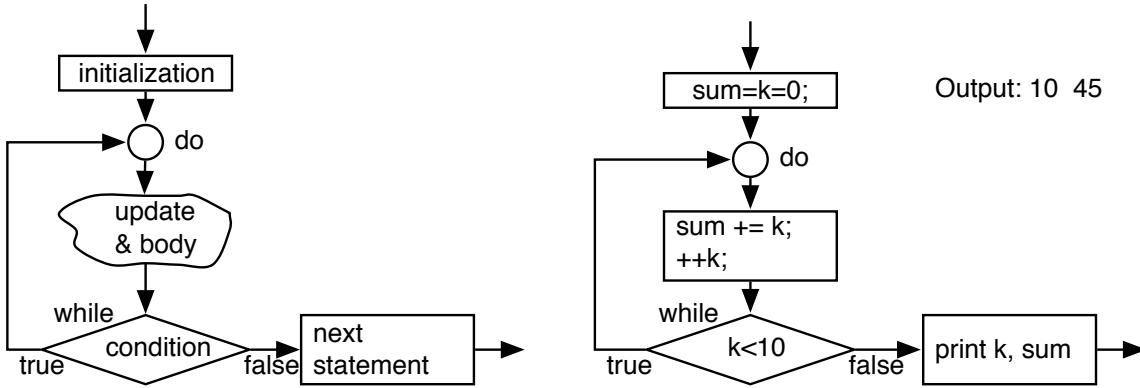


Figure 6.5. A flow diagram for the `do...while` statement.

**Updating the loop variable.** The update expression is evaluated every time around the loop, after the loop body and before re-evaluating the loop condition. Note the mismatch between *where* the update is written on the page and *when* it happens. It is written after the condition and before the body, but it happens after the body and before the condition. Beginning programmers sometimes are confused by this inverted order. Let the flow diagram be your guide to the proper order of evaluation.

In a counted `for` loop with loop variable `k`, the update expression often is as simple as `k++` or `++k`. A lot of confusion surrounds the question of which of these is correct and why. The answer is straightforward: As long as the update section is just a simple increment or decrement expression, it does not matter whether you use the prefix or postfix form of the operation. Whichever way it is written, the loop variable will be increased (or decreased) by 1 after the loop body and before the condition is retested.

**The loop body.** The loop body is executed after the condition and before the update expression. In Figure 6.1, compare the `for` loop body, which contains only one statement, with the two-statement body of the `while` loop. The update expression must be in the body of a `while` loop but becomes part of the loop header of a `for` loop, shortening the body of the loop by at least one statement. Often, this reduces the body of a `for` loop to a single statement, which permits us to write the entire loop on one line, without braces, like this:

```
for (sum = 0, k = 0; k < 10; ++k) sum += k;
```

### 6.1.2 The `do...while` Loop

The `do...while` loop implements a different control pattern than the `while` and `for` loops. The body of a `do...while` loop is executed at least once; this is illustrated by the flow diagram in Figure 6.5. The condition, written after the keyword `while`, is tested after executing the loop body. Therefore, unlike the `while` and `for` loops, the body of a `do...while` loop can initialize the variables used in the test. This makes the `do...while` loop useful for repeating a process, as shown in Figure 6.12.

### 6.1.3 Other Control Statements

C supports four statements whose purpose is to transfer control to another part of the program. Two of these, `break` and `continue`, are used in conjunction with loops to create additional structured control patterns. The third, `return`, is used at the end of a function definition. The fourth, `goto`, has little or no place in modern programming and should not be used. Neither `break` nor `continue` is necessary in C; any program can be

---

A **break** statement takes control directly out of a **while** or **do** loop.

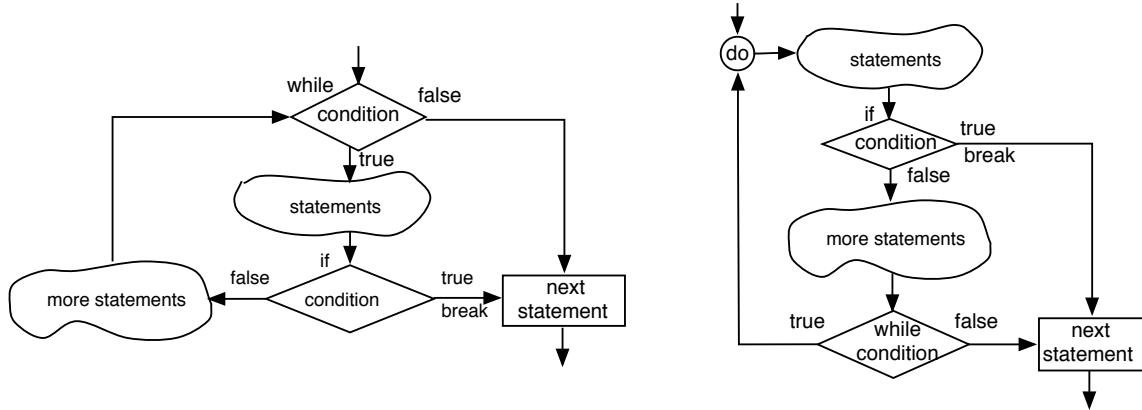


Figure 6.6. Using **break** with **while** and **do**.

written without them. However, when used skillfully, a **break** or **continue** statement can simplify program logic and shorten the code. This is highly desirable because decreasing complexity decreases errors.

**The break statement.** The **break** statement interrupts the normal flow of control by transferring control from inside a loop or a **switch**<sup>1</sup> to the statement after its end. If the **break** is within a **for**, **do...while**, or **while** loop, execution continues with the first statement following the body of the loop.

The **break** statement is diagrammed as an arrow because it interrupts the normal flow of control. An **if** statement whose true clause is a **break** statement is commonly used inside a **while** loop or a **for** loop. We will call this combination an **if...break** statement; it is diagrammed as an arrow that leaves the normal control path at an **if** statement inside the loop and rejoins the normal path at the statement after the loop. (See Figure 6.6.)

In a **for** loop, a **break** statement also takes control to the first statement after the body of the loop. This is discussed further in Section 6.2.7.

---

<sup>1</sup>Section 6.3 deals with the **switch** statement.

---

A **continue** statement takes control directly to the top of a **while** or **do** loop.

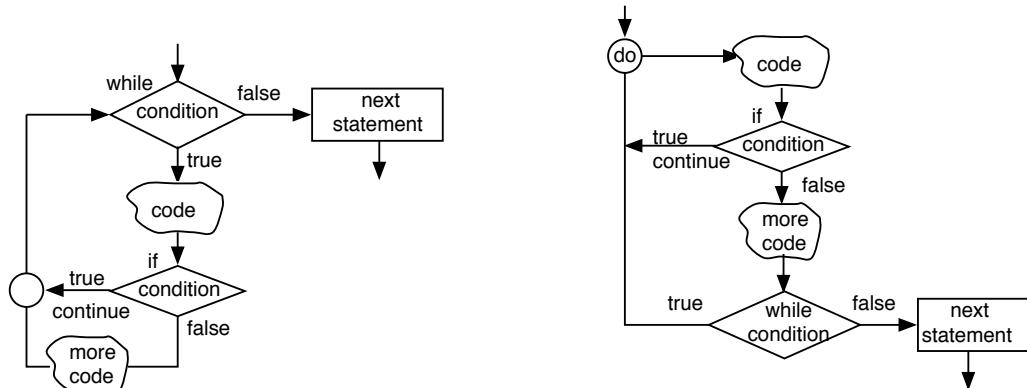
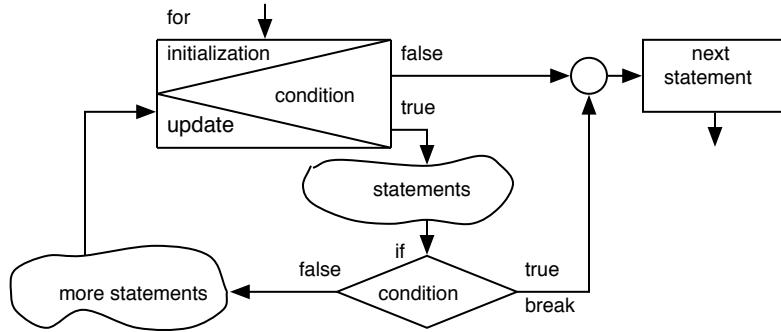


Figure 6.7. Using **continue** with **while** and **do**.

A `continue` statement takes control to the increment step in the header of a `for` loop.



**Figure 6.8. Using `continue` with `for`.**

**The `continue` statement.** The `continue` statement interrupts the normal flow of control by transferring control from inside a loop to its beginning. If it is within a `for` loop, execution continues with the increment step, as shown on the right in Figure 6.7. In a `while` or `do...while` loop, execution continues with the loop test. The `continue` statement is also diagrammed as an arrow. An `if...continue` statement is diagrammed as an arrow that leaves the normal control path at the `if` statement and rejoins the loop at the top.

The `for` loop works like a `while` loop. Figure 6.8 shows that control returns to the “update” portion of the control unit.

The `continue` statement is not commonly used but occasionally can be helpful in simplifying the logic when one control structure is nested within another, as when a loop contains a `switch` statement. This control pattern is used in menu processing and will be illustrated in Chapter 12 where `continue` is used within the `switch` to handle an invalid menu selection.

**The `return` statement.** By now, the `return` statement should be familiar to the reader; it has been used in every program example since Chapter 2. Here, we restate the rules for using `return`, and discuss a few usage options.

1. Executing a `return` statement causes control to leave a function immediately.
2. A `void` function may contain a simple `return` statement, in which the keyword `return` is followed immediately by a semicolon. If this statement is omitted, the function will return when control reaches the final closing brace.
3. Every non-void function must contain a `return` statement with an expression between the keyword `return` and the semicolon. The type of this expression must match the declared return type or be coercible to that type<sup>2</sup>. When control reaches this statement, the expression will be evaluated and its value will be returned.
4. It is syntactically legal to have more than one `return` statement in a function. However, this is generally considered to be poor style. There is genuine debate, however, about whether multiple `returns` are sometimes appropriate. On one side, many teachers and employers prohibit this practice because it breaks the “one way in – one way out” design rule. On the other side, some experts believe that code simplicity is more important than “one in – one out” construction, and permit use of multiple return statements in a few situations where the extra return statements significantly shorten or reduce the nesting level of the code.

#### 6.1.4 Defective Loops

If a loop does not update its loop variable, it will loop forever and be called an **infinite loop**. The most common cause of an infinite loop is that the programmer simply forgets to write a statement that updates the

<sup>2</sup>Type coercion is discussed in Chapter 7.

---

```

int sum = 0, k = 0; >.....prior code (initializations)....< int sum = 0, k = 0;
while (k < 10)           .....while statement.....< while (k < 10);
    sum += k;             >.....next statement.....{ sum += k;
    ++k;                  >.....next statement.....++k;
printf( "%i %i\n", k, sum );                                }
                                            }
```

---

**Figure 6.9. No update and no exit: Two defective loops.**

loop variable. Another source of error is missing or misplaced punctuation. The body of a loop starts at the right parenthesis that ends the condition. If the next character is a left curly bracket, the loop extends to the matching right curly bracket. Otherwise, the body consists of the single statement that follows the condition and the loop ends at the first semicolon. Figure 6.9 shows two loops that are infinite in nature because they do not update their loop variable. The loop on the left is like the `while` loop in Figure 6.1 except that the braces are missing. Because of this, the loop ends after the statement `sum += k;` and does not include the `++k` statement. This kind of omission will not be caught by the compiler; it is perfectly legal to write a loop with no braces and with only one statement in its body. The compiler does not check that the programmer has updated the loop variable.<sup>3</sup>

The loop on the right has an extraneous semicolon following the loop condition. This semicolon ends the loop, resulting in a null or empty loop body, which is legal in C. Since `k`, the loop variable, is not updated before this semicolon, it never changes. (The bracketed series of statements that follows the semicolon is outside the loop.) One way to avoid this kind of error is to write the left curly bracket that begins the loop body *on the same line* as the keyword `while`. The left bracket is a visible reminder that the line should not end in a semicolon.

## 6.2 Applications of Loops

Knowing the correct syntax for writing a loop is important but only part of what a programmer needs to understand. This chapter and later ones present several common applications of loops and paradigms for their implementation in C. These applications include

**Sentinel loops.** Introduced in Figure 6.24 and discussed in Section 6.2.1.

**Query loops.** Presented in Section 6.2.2.

**Counted loops.** Introduced in Figure 3.14 and treated in greater depth in Section 6.2.3.

**Input validation loops.** Introduced in Figure 3.15 and revisited in Section 6.2.4.

**Nested loops.** : Presented in Section 6.2.5.

**Delay loops.** Presented in Section 6.2.6.

**Flexible-exit loops.** Presented in Section 6.2.7.

**Counted sentinel loops.** Based on the flexible-exit loop, this pattern is presented in Section 6.2.8.

**Search loops.** Introduced in Section 6.2.9 and illustrated in several later chapters.

**Table processing loops** are introduced in Chapter 13.

---

<sup>3</sup>A mathematical technique, called a *loop invariant*, can be used to find loops that do not accomplish the design goals. This technique is beyond the scope of an introductory textbook.

*End-of-file loops* are introduced in Chapter 14.

### 6.2.1 Sentinel Loops

A **sentinel loop** keeps reading, inspecting, and processing data values until it comes across a predefined value that the programmer has designated to mean “end of data.” Looping stops when the program recognizes this value, which is called a **sentinel value**, because it stands guard at the end of the data.

The value used as a sentinel depends on the application; to choose an appropriate sentinel value, the programmer must understand the nature of the data. Most functions that process strings use the null character as a sentinel value. Loops that read and process input data often use the newline character as a sentinel. If the data values are integers and a program processes only nonzero data values, then 0 can be used as a sentinel. If all data values are nonnegative, then  $-1$  can be used as the sentinel. If every integer is admissible, the value `INT_MAX` (the largest representable integer) often is used as a sentinel.

In all cases, a sentinel value must be of the same data type as the ordinary data values being processed, because it must be stored in the same type of variable or read using the same conversion specifier in a format string. Also, a sentinel must not be contained in the set of legal data values because it must be an unambiguous signal that there are no more data sets to process.

A sentinel loop is used when the number of data items to be processed varies from session to session, depending on the user’s needs, and cannot be known ahead of time. This happens in many contexts, including

- When reading a series of input data sets.
- When processing string data.<sup>4</sup>
- When processing data that are stored in an array or a list in which the sentinel value is stored at the end of the data<sup>5</sup>.

Input-controlled sentinel loops are the only kind that we are ready to examine at this time<sup>6</sup>. An input-controlled sentinel program reads and processes input values; the sentinel value must be entered from the keyboard as a signal that there is no more input. The loop compares each input value to the sentinel and ends the input process if a match is found. Such programs follow this general form:

```
// Comments that explain the purpose of the program.
#include commands.
#define the sentinel value.

int main( void )
{
    Declaration of input variable and others.
    Output statement that identifies the program.

    Use scanf() to initialize the input variable.
    while (input != sentinel value) {
        Process the input data.
        Prompt for and read another input.
    }
    Print program results and termination comment.
    return 0;
}
```

The user prompts must give clear instructions about the sentinel value. Otherwise, the user will be unable to end the loop. For example, consider the cash register program in Figure 6.10, which uses a simple input-controlled sentinel loop. The initial prompt gives clear instructions about how to end the processing loop. Sentinel loops are implemented with `while` or `for(;;)` statements, rather than a `do...while` statement, because it is important not to try to process the sentinel value. The `do...while` statement processes every value before making the loop test, whereas the `while` loop makes the test before processing the value. Thus, a `while` loop can watch for the sentinel value and leave the loop when it appears without processing it. Figures 6.24 and 6.25 show two ways that a `for` statement can be used to implement a sentinel loop.

<sup>4</sup>Strings will be introduced in Chapter 12.

<sup>5</sup>Arrays will be introduced in Chapter 10 and lists in Chapter ??.

<sup>6</sup>The other sentinel loops will be introduced in Chapters 10, 12, and ??.

---

Compute the sum of a series of prices entered by the user.

```
#include <stdio.h>
#define SENTINEL 0

int main ( void )
{
    double input;                                // Price of one item.
    double sum = 0;                               // Total price of all items.
    printf( " Cash Register Program.\n Enter the prices; use 0 to quit.\n> " );

    scanf ( "%lg", &input );           // Read first price.
    while (input != SENTINEL) {      // Sentinel value is 0
        sum += input;                // Process the input.
        printf( "> " );              // Get next input.
        scanf( "%lg", &input );
    }

    printf( " Your order costs $%g\n", sum );
    return 0;
}
```

Output from a sample run:

```
Cash Register Program.
Enter the prices; use 0 to quit.
> 3.10
> 29.98
> 2.34
> 0
Your order costs $35.42
```

---

**Figure 6.10.** A cash register program.

### 6.2.2 Query Loops

If the loop variable is either initialized or updated by `scanf()` or some other input function, we say that it is an **input-controlled loop**. Such loops are very important because they allow a program to respond to the real-world environment. There are several variations on this theme, including query loops, sentinel loops (Section 6.2.1), and input validation loops (Section 6.2.4).

A useful interactive technique, the **repeat query**, is introduced in Figure 6.11 and used in Figure 6.12. (It will be refined, later, in Chapter 8.) To develop and debug a program, the programmer must test it with several sets of input so that its performance with different kinds of data can be checked. We use a repeat query loop to automate this process of rerunning a program. Until now, you have needed to execute a program once for each line in your test plan. After each run, the output must be captured and printed. At best, the process is awkward. At worst, the programmer is tempted to shortcut the testing process. A typical program with a query loop follows the general form shown in Figure 6.11.

The testing process can be simplified by writing a function that processes one line of the test plan. The main program contains a `do...while` loop that calls the function once and asks whether the user wishes to do it again. This provides a simple, convenient way to let the user decide whether to continue running the program or quit, rather than restarting it every time. Furthermore, all the output for all of the tests ends up in one place at one time. The basic technique is illustrated in Figures 6.11 through 6.13.

#### Notes on Figure 6.11. Form of a query loop.

- Programs that use this technique will have a few statements at the beginning of `main()` that may open files, clear the screen, or print output headings.
- At the end are statements to print final results and any closing message.

---

Many programs perform a process repeatedly, until the user asks to stop. This is the general form of such a program. The process is performed by a function called from the main loop.

```
// Comments that explain the purpose of the program.
#include and #define commands.

int main( void )
{
    Declaration of variable for query response;

    Output statement that identifies the program;
    do {
        Process one data set or call a function to do so;
        Ask the user whether to continue (1) or quit (0);
        Read the response;
    } while (response != 0);

    Print program results and termination comment;
}
```

---

**Figure 6.11. Form of a query loop.**

- In between is a `do...while` loop. The loop body consists entirely of a call on a function that performs the work of the program, followed by a prompt to ask the user whether or not to repeat the process. The response is read and immediately tested. If the user enters the code 0,<sup>7</sup> control leaves the loop. If 1 (or an erroneous response) is entered, the process is repeated.

Figure 6.14 shows a diagram of a query loop (on the left) used to repeat a calculation (the function diagrammed on the right). The dotted lines show how control goes from the function call to the entry point of the function and from the function return back to the call box.

**Notes on Figure 6.12. Repeating a calculation.** We put almost all of the program's code into a function called `work()`. The main program contains only greeting and closing messages, a loop that calls the `work()` function to do all the work, and the input statements needed to control the loop. Figure 6.14 is a flow diagram of this program.

**First box: prototypes for the two programmer-defined functions.** Two programmer-defined functions that follow the main program in the source code file are shown in Figure 6.13. The `work()` function is called from `main()`. It, in turn, calls `drop()`.

**Outer box: the main process loop.**

- Compare this version of the program to the versions in Figures 3.10 and 5.15. The input, calculations, and output have been removed from `main()` and placed in a separate function, named `work()`. These lines have been replaced by a loop that will call the `work()` function repeatedly (inner box), processing several data sets. This keeps the logic of `main()` simple and easy to follow.
- We use a loop because we expect to process a series of inputs. Since we do not know how many will be needed ahead of time, we ask the user to tell us what to do after each loop repetition.
- We prompt for a 'y' to do the process again or a 'n' to quit; the response is read into the variable `do_it_again`.
- As long as the values we read for `do_it_again` are not 'n', we call the `work()` function to read another input value and perform the computation and output processing. If the user's input is an error, that is, it is neither a 'y' nor an 'n', this program continues; it quits only if the user enters 'n'.

**Notes on Figure 6.13: The `work()` and `drop()` functions.**

---

<sup>7</sup>We use a 1 or 0 response here because it is simple. In Chapter 8, we show how to process y or n responses.

This main program consists of statements to print the program titles and a loop that will repeatedly call the `work()` function, given in Figure 6.13, until the user asks to quit. This main program can be used to repeat any process by changing the titles and the `work()` function.

```
// -----
// Determine the time it takes for a grapefruit to hit the ground when it
// is dropped, with no initial velocity, from a helicopter hovering
// at height h. Also determine the velocity of the fruit at impact.
// -----
#include <stdio.h>

void work( void );
double drop( double height );

int main( void )
{
    char do_it_again;      // repeat-or-stop switch
    puts( "\n Calculate the time it would take for a grapefruit\n"
          " to fall from a helicopter at a given height.\n" );
    do { work();
        printf( "\n Enter 'y' to continue or 'n' to quit: " );
        scanf( "%c", &do_it_again );
    } while (do_it_again != 'n');

    return 0;
}
```

**Figure 6.12.** Repeating a calculation.

**Background.** Most of the code from the `main()` program in Figure 5.15 has been moved into the `work()` function. This reduces the complexity of `main()` and makes it easy to repeat the gravity calculations with several inputs. The `work()` function contains all the declarations and code that relate to the computation. The `main()` program contains only start-up code, termination code, and the loop that calls the `work()` function.

**The flow of control.** The function call in `main()` sends control into the `work()` function, where we read one input and calculate the time of fall by calling the `drop()` function. After returning from `drop()`, we calculate the terminal velocity and print the time and velocity. Then we return to `main()`. In the flow diagram (Figure 6.14), these shifts of control are represented by dotted lines.

**The output.** Lines printed by the `main()` program are intermixed with lines from the `work()` function. Here is a sample dialog:

```
Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.

Enter height of helicopter (meters): 20
    Time of fall = 2.01962 seconds
    Velocity of the object = 19.8057 m/s

Enter 'y' to continue or 'n' to quit: 1
Enter height of helicopter (meters): 906.5
    Time of fall = 13.597 seconds
    Velocity of the object = 133.34 m/s

Enter 'y' to continue or 'n' to quit: 2
Enter height of helicopter (meters): 2000.5
    Time of fall = 20.1987 seconds
    Velocity of the object = 198.082 m/s

Enter 'y' to continue or 'n' to quit: 0
```

As you begin to write programs, incorporate a processing loop into each one. It then will be convenient for you to test your code on a variety of inputs and demonstrate that it works correctly under all circumstances.

---

The `work()` function is called from the program in Figure 6.12. The `drop()` function is a more concise version of the one in Figure 5.18.

```
// -----
// Perform one gravity calculation and print the results.
void
work( void )
{
    double h;           // height of fall (m)
    double t;           // time of fall (s)
    double v;           // terminal velocity (m/s)

    printf( " Enter height of helicopter (meters): " );
    scanf( "%lg", &h );

    t = drop( h );      // Call drop with the argument h.
    v = GRAVITY * t;    // velocity of grapefruit at impact
    printf( "     Time of fall = %g seconds\n", t );
    printf( "     Velocity of the object = %g m/s\n", v );
}

// -----
// Calculate time of fall from a given height. -----
double
drop( double height )
{
    double answer = 0;
    if (height > 0) answer = sqrt( 2 * height / GRAVITY );
    return answer;
}
```

---

Figure 6.13. The `work()` and `drop()` functions.

### 6.2.3 Counted Loops

Many loops are controlled by a counter. In such a **counted loop**, an initialization statement at the top of the loop usually sets some variable, say `k`, to 0 or 1. The update statement increments `k`, and the loop test asks whether `k` has reached or exceeded some goal value, `N`. To calculate the **trip count**, that is, the number of times the loop body will be executed,

- Let the initial value of the loop variable be  $I$  and the goal value be  $N$ .
- If the loop test has the form `k < N`, the trip count is  $N - I$ .
- If the loop test has the form `k <= N`, the trip count is  $N - I + 1$ .

The loops diagrammed in both Figures 6.4 and 6.15 are counted loops. In the first example, the loop variable is `k`, the initial value is 0, and the test is `k<10`; so this loop will be executed 10 times, with `k` taking on the values 0...9, successively. If an initial value of 1 and a loop test of `k <= N` were used, the loop body still would be executed 10 times, but the sum would be different, because `k` would have the values 1...10. Both patterns of loop control are common. A frequent source of program error is using the wrong initial value or the wrong comparison operator in the loop test.

**A summation loop.** Our next example, in Figure 6.15, shows how a counted loop can be used to sum a series of numbers. This example demonstrates a typical programming pattern in which two variables are used: a *counter* (to record the number of repetitions) and an *accumulator* (to hold the sum as it is accumulated). Both variables are initialized before the loop begins and both are changed on every trip through the loop. The flow diagram for this program (shown following the code) is very similar to that in Figure 6.4, since the same loop structure is being used.

This is a flow diagram of the program in Figures 6.12 and 6.13. The dotted lines show how the control sequence is interrupted when the function call sends control to the beginning of the function. Control flows through the function then returns via the lower dotted line to the box from which it was called.

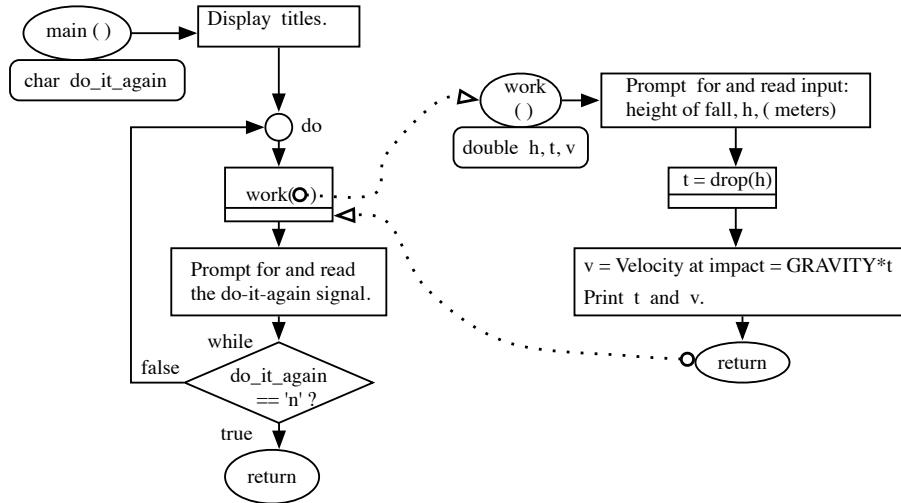


Figure 6.14. Flow diagram for repeating a process.

#### Notes on Figure 6.15. SumUp.

**First box: the function.** The same program could be used to sum a different function by simply changing the expression in the `return` statement here and the `printf()` statement in `main()`.

#### Second box: the declarations.

- The type `int` is appropriate for counters such as `n`, that are used to count the repetitions of the loop.
- An accumulator is a numeric variable used to accumulate the sum of a series of values. These might be computed values, as here, or input values, illustrated in Figure 6.10. We use a variable of type `double` for the accumulator because it will be used to compute the total of various fractions.

#### Third box: the loop.

- Before entering any loop, the variables used in the loop must be initialized. We set `n = 1` because we want to sum the series from 1 to 10. We set `sum = 0-->`.
- The `for` statement is ideally suited for counted loops because the loop header organizes all the information about where the counter starts and stops and how it changes at each step. Each time around the loop, we add  $1.0/n$  to the sum. This loop starts with `n` equal to 1 and ends when `n` reaches 11. Therefore, the loop body will be executed  $11 - 1 = 10$  times, summing the fractions  $1.0/1 \dots 1.0/10$ .
- Note that `n` will have the value 1 (not 0 or 2) the first time we compute  $1.0/n$ . This is important. First, we do not want to start computing the series at the wrong point. Second, we need to be careful to avoid dividing by 0.
- We are permitted to divide the `double` value 1.0 by the integer value `n`. The result is a fractional value of type `double`. It is important that the constant 1.0 be used rather than 1 in this expression, because the latter will give an incorrect result. The reason for this is discussed in Chapter 7.

#### Fourth box: the output.

- We use the format `%g` to print the `double` value. Up to six digits (C's default precision) will be printed with the result rounded to the sixth place. Trailing 0's, if any, are suppressed after the decimal point.
- The output from this program is

This program computes the sum of the first  $N$  terms of the series  $1/n$ .

```
#include <stdio.h>
#define N 10

double f( double x ) { return 1.0 / x; }      // The function to sum.

int main( void )
{
    int n;                      // Loop counter
    double sum;                  // Accumulator

    printf( "\n Summing 1/n where n goes from 1 to %i \n", N );
    sum = 0;                     // Start accumulator at 0.
    for (n = 1; n <= N; ++n) {   // Sum series from 1 to N.
        sum += f( n );
    }

    printf( " The sum is %g.\n", sum );
    return 0;
}
```

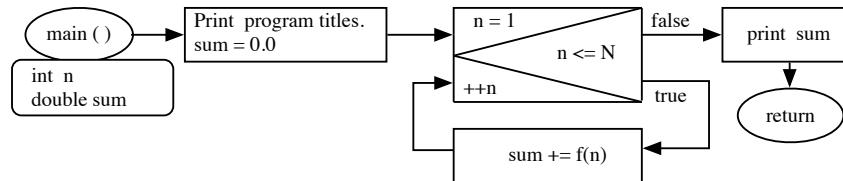


Figure 6.15. Summing a series.

```
Summing 1/n where n goes from 1 to 10
The sum is 2.92897.
```

#### 6.2.4 Input Validation Loops

Figure 6.16 contains a validation loop based on a `while` statement. It provides good user feedback but is long and requires duplicating lines of code. We can write a shorter, simpler validation loop that uses a `do...while` statement. However, this kind of validation loop has a severe defect: It gives the same prompt for the initial input and for re-entry after an error. This is a human-engineering issue. The error may go unnoticed if the program does not give distinctive feedback. A third kind of validation loop can be written with `for` and `if...break` statements that combines the advantages of the other two forms; it avoids duplication of code and provides informative feedback when the user makes an error (see Figure 6.15).

##### Notes on Figure 6.16: Input validation using a while statement.

1. The input prompt `scanf()` and calculation statements are written before the loop and again in the body of this loop. This duplication is undesirable because both sets of statements must be edited every time the prompt or input changes. The duplication is unavoidable, though, because the loop variable, `hours`, must be given a value before the `while` test at the top of the loop. It then must be given a new value within the loop.
2. The `while` test checks whether the value of `hours` is within the legal range. If not, we enter the body of the loop, print an error comment, and prompt for and read another input. Then control returns to the

---

This input validation loop is a fragment of the miles-per-hour program in Figure 3.15. Some output from the loop and the following `printf()` statement follow.

```
printf( " Duration of trip in hours and minutes: " );
scanf( "%lg%lg", &hours, &minutes );
hours = hours + ( minutes / 60 );
while (hours < 0) {
    printf( " Please re-enter; time must be >= 0: " );
    scanf( "%lg%lg", &hours, &minutes );
    hours = hours + ( minutes / 60 );
}
```

Output:

```
Duration of trip in hours and minutes: -148 43
Please re-enter; time must be >= 0: 1 -70
Please re-enter; time must be >= 0: 148 -17
Average speed was 1.94291
```

---

**Figure 6.16. Input validation using a while statement.**

top of the loop to test the data again.

3. Since only a legal input will get the user out of this loop, an informative prompt is very important. If the user is not sure what data values are legal, the program may become permanently stuck inside this loop and it may be necessary to reboot the computer to regain control.

### 6.2.5 Nested Loops

A general rule of programming is that function follows form, that is, the form or shape of the input or output data frequently determines the way that code is written to process it. This is never more true than when processing tables. The rows and columns of a table are reflected in the code in the form of a loop to process the columns written within a loop that process the rows. We call such a control structure a *nested loop*<sup>8</sup>. This control structure is illustrated in a simple but general form by the program in Figure 6.17, which prints a 10-by-12 multiplication table. Its flow diagram is given in Figure 6.18.

**Notes on Figure 6.17. Printing a table with nested for loops.** This program prints an  $R$ -by- $C$  multiplication table, where  $R$  and  $C$  are #defined as 10 and 12.

**Outer box: The row loop.** The outer loop is executed 10 times, once for each row of the table. Its body consists of the code to process one row; it prints a row label, processes all columns (the inner loop), and finishes the row by printing a newline and a vertical bar on the next line. This is a very typical processing pattern for a nested loop that does output. The output from one repetition, one row of numbers followed by a blank row looks like this:

```
1. | 1 2 3 4 5 6 7 8 9 10 11 12
```

**Inner box: the column loop.** The output from each repetition of the inner loop is one column of one row of the table, consisting of two or three spaces and a number. The `printf()` in the inner loop will be executed 12 times per trip through the outer loop. After the 12th number has been printed, the inner loop exits and control goes to the `printf()` at the end of the outer box.

---

<sup>8</sup>For processing two-dimensional arrays, a `for` loop within a `for` loop is the dominant control pattern. This will be fully explored in Chapter 18.

This program prints a multiplication table with 10 rows and 12 columns. The line number and a vertical line are printed along the left margin.

```
#include <stdio.h>
#define R 10
#define C 12

int main( void )
{
    int row, col;

    banner();
    printf( "\n\n Multiplication Table \n\n" );

    for (row = 1; row <= R; ++row) {           // Print R rows.
        printf( "%2i. |", row );                // Print left edge of row.

        for (col = 1; col <= C; ++col) {        // Print C columns in each row.
            printf( "%4i", row * col );
        }

        printf( "\n      |\n" );                  // Print blank row.
    }

    printf( "\n\n" );
    return 0;
}
```

Figure 6.17. Printing a table with nested for loops.

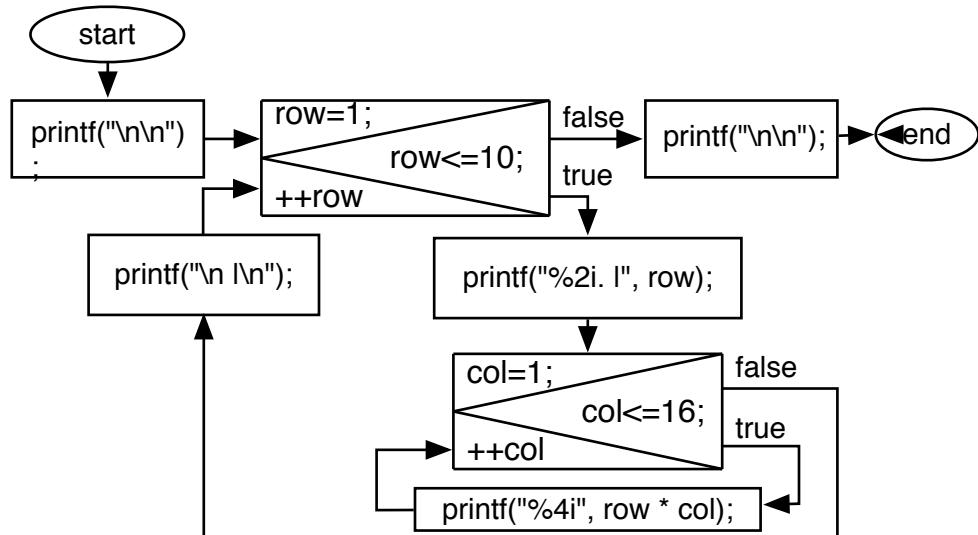


Figure 6.18. Flow diagram for the multiplication table program.

**After 120 trips.** In this program, control goes through the outer loop 10 times. For each trip through the outer loop, control goes through the inner loop 12 times. Therefore, control passes through the body of the inner loop (a `printf()` statement) a total of 120 times, processing every column in every row of the table. After finishing the 10th row, control goes to the final `printf()` and the `return` statement. The complete output is

Multiplication Table												
1.	1	2	3	4	5	6	7	8	9	10	11	12
2.	2	4	6	8	10	12	14	16	18	20	22	24
3.	3	6	9	12	15	18	21	24	27	30	33	36
4.	4	8	12	16	20	24	28	32	36	40	44	48
5.	5	10	15	20	25	30	35	40	45	50	55	60
6.	6	12	18	24	30	36	42	48	54	60	66	72
7.	7	14	21	28	35	42	49	56	63	70	77	84
8.	8	16	24	32	40	48	56	64	72	80	88	96
9.	9	18	27	36	45	54	63	72	81	90	99	108
10.	10	20	30	40	50	60	70	80	90	100	110	120

### 6.2.6 Delay Loops

A loop that executes many times but does nothing useful can be used to make the computer wait for a while before proceeding. Such a loop is called a **delay loop**. Delay loops often are used like timers to control the length of time between repeated events (see Figure 6.19). For example, a program that controls an automated factory process might use a delay loop to regulate sending analog signals to (or receiving them from) a device such as a motor generator.

**Notes on Figures 6.20 and 6.19. Delaying progress, the `delay()` function.** The `delay()` function implements a delay loop (see Figure 6.20). It calls the C library function `time()` to read the computer's real-time clock and return the current time, in units of seconds, represented as an integer so that we can do arithmetic with it. The type `time_t` is defined<sup>9</sup> by your local C system to be the right kind of integer<sup>10</sup> for storing the time on your system.

We add the desired number of seconds of delay to the current time to get the goal time, then store it in the variable `goal`. The loop calls `time()` continuously until the current time reaches the goal time. This loop is all test and no body. Technically, it is called a **busy wait** loop because it keeps the processor busy while waiting for time to pass. It is busy doing nothing, that is, wasting time<sup>11</sup>. On a typical personal computer, the `time()` function might end up being called 100,000 times or more during a delay of a few seconds. Busy waiting is an appropriate technique to use when a computer is dedicated to monitoring a single experiment or process. It is not a good technique to use on a shared computer that is serving other purposes simultaneously.

A delay loop usually is used inside another loop, which must perform a process repeatedly at a particular rate that is compatible with human response or a process being monitored. For example, the boxed loop in Figure 6.19 is used to time repetitions of an exercise. It outputs `\a` (a beep), then calls `delay()`, which waits the number of seconds specified by the user before returning. The output looks like this:

```
This is an exercise program.
```

```
How many pushups are you going to do? 5
How many seconds between pushups? 3
OK, we will do 5 pushups, one every 3 seconds.
Do one pushup each time you hear the beep.
1
2
```

<sup>9</sup>Type definitions are discussed in Chapters 8, 12, 13, and 18.

<sup>10</sup>The various kinds of integers are discussed in Chapter 7.

<sup>11</sup>This is legal in C; a loop is not required to have any code in its body.

A delay loop is used here to regulate repetitions of a process.

```
#include <stdio.h>
void delay( int seconds );
int main( void )
{
    int j, max, seconds;
    printf( "This is an exercise program.\n\n"
            "How many pushups are you going to do? " );
    scanf( "%i", &max );
    if (max < 0) {
        printf( "Can't do %i pushups!\n", max );
        exit( 1 );
    }
    printf( "How many seconds between pushups? " );
    scanf( "%i", &seconds );
    if (seconds < 3) {
        printf( "Can't do a pushup in %i seconds!\n", seconds );
        exit( 1 );
    }
    printf( "OK, we will do %i pushups, one every %i seconds.\n",
            "Do one pushup each time you hear the beep.\n", max, seconds );
    for (j = 1; j <= max; ++j) {
        printf( "%i \a\n", j );      // Do one.
        delay( seconds );          // Wait specified # of seconds.
    }
    puts( "Good job. Come again." );
}
```

**Figure 6.19.** Using a delay loop.

---

You may wish to put this function in your personal `mytools` library.

```
#include <time.h>
void delay( int seconds )
{
    time_t goal = time( NULL ) + seconds;    // Add seconds to current time.
    do {      // Nothing      } while (time( NULL ) < goal);
}
```

**Figure 6.20.** Delaying progress, the `delay()` function.

---

The only exit from this loop is through the **break** statement. The **for** header is used to initialize and update a counter but it has no exit test.

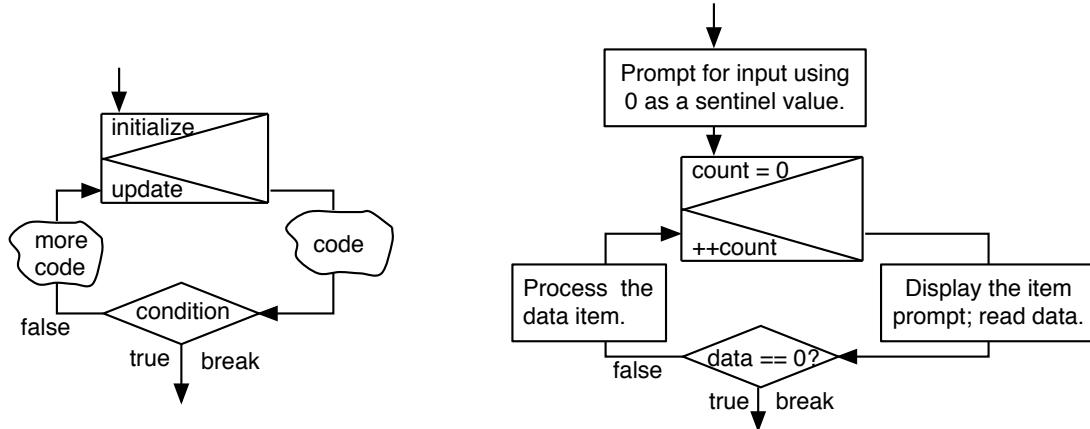


Figure 6.21. A structured loop with break.

```

3
4
5
Good job. Come again.

```

### 6.2.7 Flexible-Exit Loops

Some languages support a kind of loop that permits the programmer to place the loop test anywhere between the beginning and the end of the loop body. Such a loop takes the place of the **while** and **do...while** loops in C and also provides other options. At the same time, it remains a one-in, one-out control structure, and therefore, is consistent with “structured programming”. This sort of flexible-exit structured loop can be imitated in C by a combination of three control statements: a **for** statement with empty loop test<sup>12</sup>, where the only exit is by way of an **if** statement and a **break** statement somewhere within the loop body<sup>13</sup>. The skeleton of a **for** loop with an **if...break** is shown below and diagrammed on the left in Figure 6.21.

```

for (initialization; ; update) {
    statements
    if (condition) break;
    more statements
}
next statement // Control comes here after the break.

```

This degenerate statement sometimes is called an **infinite for loop** because the loop header has no test that can end the execution of the loop. A real infinite loop is not particularly useful because it never ends. However, an infinite **for** loop normally contains an **if...break** statement and, therefore, is not infinite because the **if...break** provides a way to leave the loop. Applications of the infinite **for** loop are shown in Figure 6.22, where it is used for data validation, and in Figure 12.37, where an infinite loop is combined with a **switch** to implement a complex control structure.

A loop that exits by means of an **if...break** statement has one big advantage over an ordinary loop: the loop body can have some statements before the loop test and more statements after it. This flexibility makes it easier to write code in many situations: it provides a straightforward, clear, nonredundant way to do jobs that are awkward when done with other loops. Its primary application is in loops that depend on some property of data that is being read and processed. The first part of the loop does the input or calculation, followed immediately by the test that stops the looping when data with a specified property is found.

<sup>12</sup>It is legal to omit one, two, or all of the expressions in the loop header. However, the two semicolons must be present.

<sup>13</sup>Some practitioners use **while (1)** or **while (true)** instead of **for (;;)**. However, B. Kernighan, one of the inventors of C, wrote to me that he prefers the **for(;;)**, possibly because it avoids writing a meaningless test condition.

---

Compare this input validation loop to the version in Figure 6.16 that is built on the `while` statement. This form is simpler and avoids duplication. The output is identical to the output from Figure 6.16.

```
printf( " Duration of trip in hours and minutes: " );
for (;;) {
    scanf( "%lg%lg", &hours, &minutes );
    hours = hours + ( minutes / 60 );
    if (hours >= 0) break;           // Leave loop if input is valid.
    printf( " Please re-enter; time must be >= 0: " );
}
```

---

**Figure 6.22.** Input validation loop using a `for` statement.

**Notes on Figure 6.22. Input validation using a `for` statement.** Compare the code in Figure 6.22 to the validation loop in Figure 6.16.

1. The original input prompt is written before the loop because it will not be repeated. (The error prompt is different.)
2. This loop can be written using a `while` statement as in Figure 6.16. However, since the `while` test is at the top of the loop, this implementation requires the input and calculation statements to be written twice, once before the loop and once in the loop body.
3. In contrast, if we use a flexible-exit loop, there is no need to write the `scanf()` and computation twice. The resulting code is simpler and clearer.
4. The input and calculation statements are done before the loop test, which is in an `if...break` statement. If the input is valid, control leaves the loop.
5. If not, we print an error prompt, return to the top of the loop, and read new data.
6. Since only a legal input will get the user out of this loop, an informative prompt is very important. If the user is not sure what data values are legal, the program may become permanently stuck inside this loop, and it may be necessary to reboot the computer to regain control.

### 6.2.8 Counted Sentinel Loops

Earlier in this section, we discussed a sentinel loop based on `while`. Often, we combine the sentinel test with a loop counter to make a counted sentinel loop. The general design for such a loop is given on the left in Figure 6.23. In this design, `LIMIT` is the maximum number of times the loop should be repeated, and `SENTINEL` is the designated sentinel value. To illustrate this pattern, we add a counter to the cash register program from Figure 6.10. The improved program is shown in Figure 6.24.

---

Two designs are given for a counted sentinel loop. The version on the left is simpler. The version on the right avoids use of `break`.

<pre>int done = 0; // false for (k = 0; k &lt; LIMIT; ++k) {     Prompt for and read an input.     if (input == SENTINEL) break;     Process the input data. }</pre>	<pre>for (k = 0; k &lt; LIMIT &amp;&amp; !done; ++k) {     Prompt for and read an input.     if (input == SENTINEL) done = 1; // true     else {         Process the input data.     } }</pre>
--	--

---

**Figure 6.23.** Skeleton of a counted sentinel loop.

---

We use a **for** loop to count the number of data items that are entered, and a **break** statement to leave the loop when the input is a designated sentinel value.

```
#include <stdio.h>
#define SENTINEL 0 // Signal for end of input.
int main( void )
{
    double input; // Price of one item.
    double sum; // Total price of all items.
    int count; // Number of items.

    puts( " Cash Register Program.\n"
          " Enter prices; 0 to quit." );
    for (count=sum=0; ;++count) {
        printf( "--> " );
        scanf( "%lg", &input );
        if (input == SENTINEL) break;
        sum += input;
        printf( "\t Input: %g\n", input );
    }
    printf( " Your %i items cost $%g\n", count, sum );
    return 0;
}
```

The output is :

```
Cash Register Program.
Enter prices; 0 to quit.
--> 3.17
Input: 3.17
--> 2.35
Input: 2.35
--> 0.78
Input: 0.78
--> 10.52
Input: 10.52
--> 0
Your 4 items cost $16.82
```

---

Figure 6.24. Breaking out of a loop.

---

This program uses a status flag (**done**) instead of a **break** instruction to leave the loop. Compare it to the simpler logic of Figure 6.24, that relies on the **break**.

```
#include <stdio.h>
#define SENTINEL 0 // Signal for end of input.
int main( void )
{
    double input; // Price of one item.
    double sum; // Total price of all items.
    int count; // Number of items.
    int done = 0; // Set to 1 when sentinel is entered.

    puts( " Cash Register Program.\n Enter prices; 0 to quit." );
    for (count=sum=0; !done ;++count) {
        printf( "--> " );
        scanf( "%lg", &input );
        if (input == SENTINEL) done = 1;
        else {
            sum += input;
            printf( "\t Input: %g\n", input );
        }
    }
    printf( " Your %i items cost $%g\n", count-1, sum );
    return 0;
}
```

---

Figure 6.25. Avoiding a break-out.

---

Two designs are given for a search loop, with and without the use of `break`.

```
Read or select the key value.
for (k = 0; k < LIMIT; ++k) {
    Calculate / select next item to test.
    if (current_data == key_value) break;
}

int done = 0; // false
Read or select the key value.
for (k = 0; k < LIMIT && !done; ++k) {
    Calculate / select next item to test.
    if (current_data == key_value) done = 1;
}
```

---

**Figure 6.26.** Skeleton of a search loop.

**Moving the exit test to the top.** A counted sentinel loop can be written two ways, with and without the use of `break`. Some professionals believe that the `break` statement should never be used because it is possible to overuse `break` and use it as a substitute for clean logic design. It is possible to implement the flexible-exit loop without the `break` statement by adding only a few lines of code. The loop design on the right of Figure 6.23 shows the kind of additions that are necessary to avoid the `break`.

We have added a status variable, `done`, that is set to false initially and then to true when the designated sentinel value is read. The “more code” section of the loop body is enclosed in an `else` clause so that the sentinel value will not be processed. Then the status variable is tested again and the loop ends. This is slightly less efficient and slightly longer to write than the version that uses `break`. Note, also, that the counter will be incremented one extra time and, therefore, we must subtract one from its value to get the true number of items that have been processed. The program in Figure 6.25 uses this logic.

### 6.2.9 Search Loops

A **search loop** examines a set of possibilities, looking for one that matches a given “key” value. The data items being searched can be stored in memory or calculated. The key value could be the entire item or part of it and it could be of any data type. The requirements for searching are almost identical in all cases:

- The program must know what key value to look for.
- There must be some orderly way to examine the possibilities, one at a time, until all have been checked.
- The loop must compare the key value to the current possibility. If they match, control must leave the loop.
- The search loop must know how many possibilities are to be searched or have some way to know when no possibilities remain. The loop must end when this occurs.

Therefore, the search loop will terminate for two possible reasons: The key item has been found or the possibilities have been used up. The most straightforward way to implement this control pattern is to use a counted sentinel loop, with either the status flag or the `break` statement. The general pattern of a search loop is shown in Figure 6.26.

A search loop based on data input is shown in Figure 6.32. Search loops based on computed possibilities are illustrated in Figures ?? and in the Newton’s method program on the text website. Search loops become increasingly important when there are large amounts of stored data. Loops that search arrays are illustrated in Chapter 18.

## 6.3 The `switch` Statement

C provides three ways to make a choice: the `if...else` statement, the `switch` statement, and the conditional operator. The `if...else` was fully covered in Chapter 3, this section is devoted to the `switch`, and the conditional operator is explained in Appendix D.

### 6.3.1 Syntax and Semantics

A `switch` implements the same control pattern as a series of `if...else` statements. (Refer to Figure 3.11.) In both, we wish to select one action from a set of possible actions. A `switch` could be thought of as a shorter,

integrated notation for the same logic as a nested `if...else` statement. Normally, one would use `if...else` when there are only two alternatives; a `switch` is only helpful when there are more than two choices.

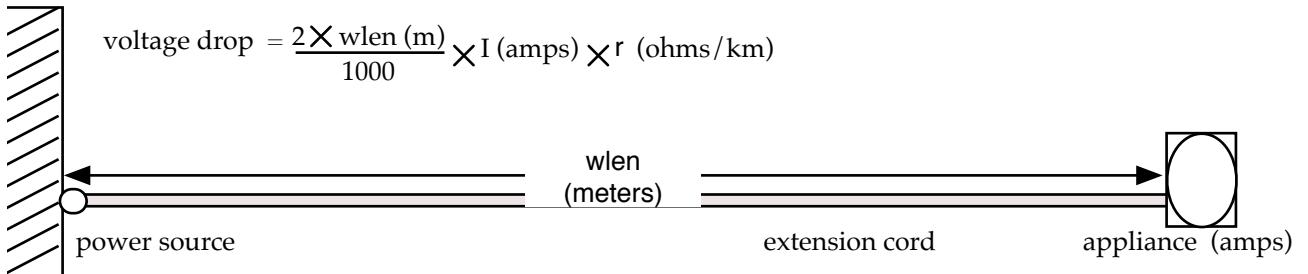
**The syntax of a switch statement.** To illustrate the syntax and use of the `switch` statement, we use a program whose specification is given in Figure 6.27 and program code in Figure 6.29. A `switch` statement has several parts, in this order:

1. The keyword `switch`
2. An expression in parentheses that computes a value of some integral type<sup>14</sup>

<sup>14</sup>C has several integral types in addition to `int`; they will be introduced in the next few chapters. These include `char`, `short`,

**Problem scope:** A contractor wants to automate the process of selecting an appropriate gauge wire for extension cords and temporary wiring at construction sites. Various long wires are used to supply electricity to power tools and other appliances at the site. All wires are standard annealed copper; calculations are to be made for a standard temperature of 20°C. There is a voltage drop in an extension cord due to the resistivity of its wire; heavier cords (lower-numbered gauges) have lower resistivity and incur less drop than lighter cords (higher-numbered gauges). The voltage drop is proportional to the length of the wire, so the drop can be significant in a long wire. This is an issue because appliances designed to operate at one voltage may overheat if operated at a voltage that is significantly lower. This program should evaluate a proposed wiring scheme and answer whether the wire will be adequate for its intended purpose.

**Formula:** An extension cord  $n$  meters long contains  $2n$  meters of wire. The voltage drop is



where  $wlen$  is the length of the extension cord (in meters),  $\rho$  (rho) is the resistivity of the wire, and  $I$  is the current flowing in the wire, in amperes.

**Constants:** A voltage drop of up to 5 volts is acceptable. The resistivity,  $\rho$ , of copper wire at 20°C for the gauges used by this contractor are

Gauge	$\rho$
12	5.211
14	8.285
16	13.17
18	20.95

**Inputs:** The contractor will type in the length of the extension cord he needs, the wire gauge, and the current rating (amps) of the appliance he will be using. The program should reject any gauge that is not in the table above.

**Output required:** Program headings and a termination message should be printed. Input values should be echoed. The voltage drop and the answer (gauge is adequate or not adequate) should be given. Three digits of precision are adequate.

Figure 6.27. Problem specification: Wire gauge adequacy evaluation.

We diagram a series of `if...else` statements that implements the specification in Figure 6.27.

We diagram the same logic again using a `switch` statement instead of an `if...else` statement.

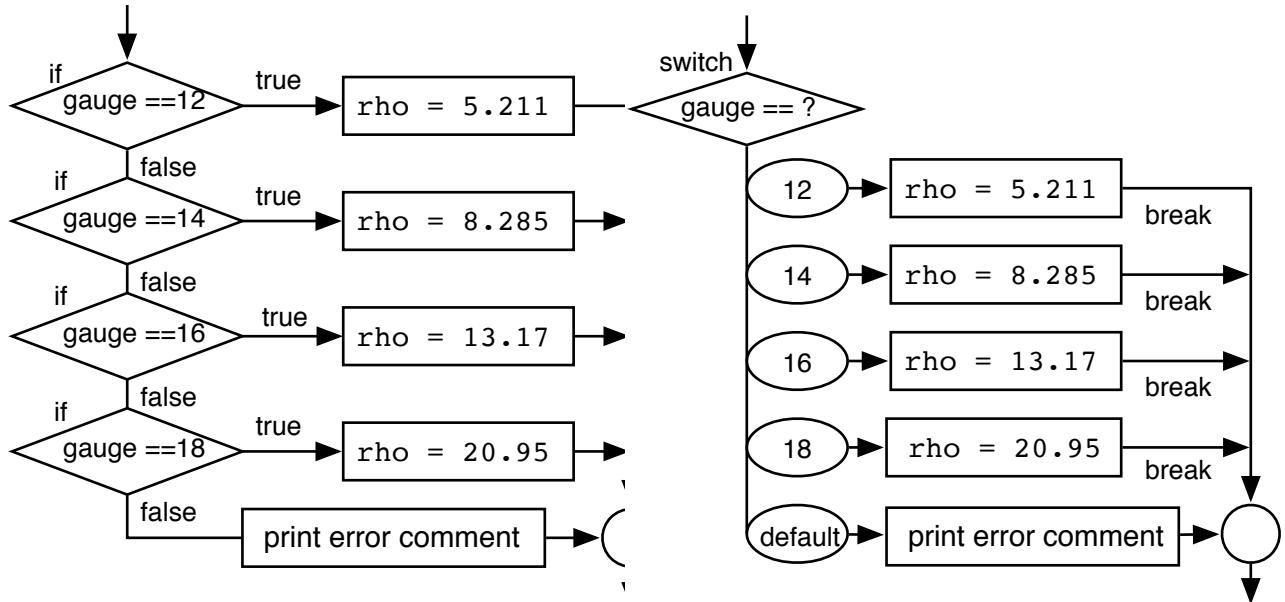


Figure 6.28. Diagrams for a nested conditional and a corresponding `switch`.

3. Braces enclosing a series of labeled `cases`. The case labels must be constants or constant expressions<sup>15</sup> of the same type as (2).
4. Each case contains a series of statements. The last statement in the series is usually, but not always, a `break` statement.
5. One of the cases may be labeled `default`. If a `default` case is present, it is traditional (but not necessary) to place it last.
6. If several cases require the same processing, several case labels may be placed before the same statement. The last label may be `default`.

**Execution of a `switch` statement.** When a `switch` is executed, the expression in (2) is evaluated and its value compared to the case labels. If any case label matches, control will go to the statement following that label. Control then proceeds to the following statement and through all the statements after that until it reaches the bottom of the `switch`. This is *not* normally what a programmer wishes to do. It is far more common to want the cases to be mutually exclusive. This is why each group of statements normally ends with a `break` statement that causes control to skip around all the following cases and go directly to the end of the `switch` statement. Programmers sometimes absentmindedly forget a `break` statement. In this case, the logic flows into the next case below it instead of to the end of the `switch` statement. Remember, this is not an error in the eyes of the compiler and it will not cause an error comment.

If no case label matches the value of the expression, control goes to the statement following the `default` label. If there is no `default` label, control just leaves the `switch` statement. This does not cause an error at either compile time or run time.

**Diagramming a `switch` statement.** The diagram of a nested `if...else` statement has a series of diamond-shaped `if` boxes, each enclosing a test, as shown on the left in Figure 6.28. The `true` arrow from each box leads to an action and the `false` arrow leads to the next `if` test. These tests will be made in sequence until some result is `true`.

<sup>15</sup>long, unsigned short, unsigned int, and unsigned long.

<sup>15</sup>A **constant expression** is composed of operators and constant operands.

In contrast, the diagram of a `switch` statement has a single diamond-shaped test box with a branching “out” arrow. This box encloses an expression whose value will be compared to the case labels. One branch is selected and the corresponding actions in the body of the `switch` statement are executed. Each set of actions must end in a `break` statement. Control normally enters a box from the case label and leaves by an arrow that goes directly to the connector at the end of the `switch` statement. The diagram on the right of Figure 6.28 shows the `switch` statement that is used in Figure 6.29. Compare this to the `if...else` to its left. They implement the same logic; however, the version that uses `switch` is simpler.

### 6.3.2 A switch Application

Figure 6.29 shows a program that implements this `switch` statement to solve the problem specified in Figure 6.27. It illustrates “messy” integer case labels; that is, they are not consecutive numbers starting at 0 or 1<sup>16</sup>.

#### Notes on Figure 6.29: Using a switch.

**First box: input for the switch.** We display a list of available gauges and prompt the user for a choice. The user sees this set of choices:

```
Wire Gauge Adequacy Evaluation

Please choose gauge of wire:
  12 gauge
  14 gauge
  16 gauge
  18 gauge
Enter selected gauge:
```

#### Second box: the switch statement.

- A `switch` that processes an integer input must have integer constants for case labels. This `switch` has four cases to process the four gauges plus a `default` case for errors.
- Each “correct” case contains one assignment statement that stores the resistivity value for the selected gauge wire. Each assignment is followed by a `break` that ends the case. A `default` case does not need a `break` because it is always last.
- If more extensive processing is needed, a program might call a different function to process each case.
- Error handling is done smoothly in this program. The `default` case intercepts inputs that are not supported and calls `fatal()` to print an error comment and abort the program. By calling `fatal()`, we avoid printing meaningless answers. An example would be

```
Enter selected gauge: 11
Gauge 11 is not supported.

WireGuage has exited with status 1.
```

#### Third box: calculating the voltage drop.

- Control goes to this box after every `break`. It does not go here after executing the `default` clause because that clause calls `fatal()`, which aborts execution.
- This box prompts for and reads the rest of the input data. This is done after the `switch` statement, not before, because the menu selection can be an invalid choice and there is no point reading the rest of the data until we know the gauge is one of those listed in the table.
- We calculate the voltage drop as soon as all the data have been read. The formula for voltage drop uses the resistivity value selected by the `switch` statement. We divide by 1,000 because  $\rho$  is given in ohms/kilometer and the wire length is given in meters. We multiply by 2 because each extension cord contains a pair of wires running its full length.

---

<sup>16</sup>More applications of the `switch` statement are shown in Figures ??, 12.34, and 15.27.

---

The specifications for this program are in Figure 6.27.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXDROP 5.0      // volts

int main( void )
{
    int gauge;          // selected gauge of wire
    double rho;          // resistivity of selected gauge of wire
    double amps;         // current rating of appliance
    double wlen;         // length of wire needed
    double drop;          // voltage drop for selected parameters

    printf( "\n Wire Gauge Adequacy Evaluation" );

    printf( "\n Please choose gauge of wire:\n"
            "\t 12 gauge \n\t 14 gauge \n"
            "\t 16 gauge \n\t 18 gauge \n"
            " Enter selected gauge: " );
    scanf( "%i", &gauge );

    switch (gauge) {
        case 12: rho = 5.211; break;
        case 14: rho = 8.285; break;
        case 16: rho = 13.17; break;
        case 18: rho = 20.95; break;
        default: printf( " Gauge %i is not supported.\n", gauge );
                  exit(1);
    }

    printf( " Enter current rating for appliance, in amps: " );
    scanf( "%lg", &amps );
    printf( " Enter the length of the wire, in meters: " );
    scanf( "%lg", &wlen );
    drop = 2 * wlen / 1000 * rho * amps ;

    printf( "\n For %i gauge wire %g m long and %g amp appliance,\n"
            " voltage drop in wire = %g volts. (Limit is %g.) \n"
            "gauge, wlen, amps, drop, MAXDROP );

    if (drop < MAXDROP)
        printf( "\n Selected gauge is adequate.\n" );
    else
        printf( "\n Selected gauge is not adequate.\n" );

    return 0;
}
```

---

Figure 6.29. Using a switch.

1. **Problem scope:** Write a program to play an interactive guessing game with the user. The user is given a fixed number of guesses to find a hidden number.
2. **Constants:** The hidden number will be between 1 and 30; the user will be given up to 5 guesses.
3. **Inputs:** The user will enter a series of guesses.
4. **Output required:** The program will respond each time by saying the guess is too low, correct, or too high. If the guess is correct, the program should display the message “you win”. If the available guesses are used up, the program should display the message “You lose”.
5. **Other:** An input value outside of the specified range will be counted as a wrong guess. If the user makes optimal guesses, he can always win.

---

**Figure 6.30. Problem specification: Guess my number.**

**Fourth box: the answers.** We ran the program and tested two cases. Each time the program was run, the greeting comment, menu, and termination comment were printed; for brevity, these are not repeated here. Dashed lines are used to separate the runs.

```

Enter selected gauge: 12
Enter the current rating for the appliance, in amps: 10
Enter the length of the wire, in meters: 30

For 12 gauge wire 30 meters long and 10 amp appliance,
voltage drop in wire = 3.1266 volts. (Limit is 5 volts.)

Selected gauge is adequate.

-----
Enter selected gauge: 16
Enter the current rating for the appliance, in amps: 10
Enter the length of the wire, in meters: 30

For 16 gauge wire 30 meters long and 10 amp appliance,
voltage drop in wire = 7.902 volts. (Limit is 5 volts.)

Selected gauge is not adequate.

```

## 6.4 Search Loop Application: Guess My Number

The program specified in Figure 6.30 and shown in Figure 6.31 is a simple interactive game in which the player is given a limited number of turns to guess (and enter) the hidden number. The game ends sooner if the player’s input equals the program’s hidden number (the sentinel value). The implementation uses a counted sentinel loop with `break`. The `for` statement is used to count the player’s guesses and the `if...break` is used in the usual way to implement a possible early exit after a correct guess. Figure 6.32 is a flow diagram for this program<sup>17</sup>.

**Notes on Figures 6.32 and 6.31: A sentinel loop using for and if...break statements.**

**First box, Figure 6.31: the concealed number.** This program is a simplification of an old game that asks the user to guess a concealed number. The computer responds to each guess by telling the user whether the guess was too large, too small, or right on target. In a complete program, the concealed number would be randomly chosen and the number of guesses allowed would be barely enough (or not quite enough) to win the game every time. We give a full version of this program in Chapter 7. In this simplification, we arbitrarily choose 17 as the concealed number. The five guesses allowed are enough to uncover this number if the user makes no mistakes.

**Outer box: operation of the loop.**

- This `for` loop prompts for, reads, checks, and counts the guesses. It will allow the user up to five tries to enter the correct number. On each trial, the program gives feedback to guide the user in making the next guess. After the fifth unsuccessful try, the loop test will terminate the loop.
- We initialize the loop counter to 1 (rather than the usual 0) because we want to print the counter value as part of the prompt. The computer does not care about the counter values, but users prefer counters that start at 1, not 0.

---

<sup>17</sup>We will revisit and elaborate on this game in Chapter 7.

---

This program illustrates the simplest form of search loop: we search the input data for a value that matches the search key. The implementation uses a counted sentinel loop with `break`.

```
#include <stdio.h>
#define TRIES 5
int main( void )
{
    int k = 0;           // Loop counter.
    int guess;          // User's input.
    int num = 17;

    printf( " Can you guess my number? It is between 1 and 30.\n"
            " Enter a guess at each prompt; You have %i tries.\n", TRIES );
    for (k = 1; k <= TRIES; ++k) {
        printf( "\n Try %i: ", k );
        scanf( "%i", &guess );
        if (guess == num) break;
        if (guess > num) printf( " No, that is too high.\n" );
        else printf( " No, that is too low.\n" );
    }
    if (guess == num) printf( " YES!! That is just right. You win! \n" );
    else printf( " Too bad --- You lose again!\n" );
    return 0;
}
```

---

**Figure 6.31.** An input-driven search loop.

- We use `<=` to compare the loop counter to the loop limit because the loop variable was initialized to 1, not 0, and we want to execute the loop when the counter equals the number of allotted trials.
- Each guess has three possibilities: It can be correct, too low, or too high. To check for three possibilities, we need two `if` statements. The first `if` statement is in the inner box. If the input matches the concealed number, the `break` is executed. Control will leave the `for` loop and go to the first statement after the loop. The second `if` statement prints an appropriate comment depending on whether the guess is too high (the true clause) or too low (the false clause). Control then goes back to the top of the loop.

**Inner box:** There are two ways to leave the loop: either the guess was correct or the guesser was unable to find the hidden number in the number of tries allowed. Here, we test the number of guesses that were used to distinguish between these two cases, then print a success or failure comment. This is a typical way to handle a loop with two exit routes.

**The diagram: Figure 6.32.**

- A `break` statement is represented by an arrow and a connector, not by a rectangle, diamond, or ellipse. Note the word `break` on the `true` arrow of the `if` condition diamond. The circular connector on the loop's exit arrow is for the `break`.
- The `if...break` statement is the first diamond in the `for` loop. If the guess is not correct, control stays in the loop and enters the `if...else` statement at the end of the loop body. If the guess is correct, control flows out of the loop along the `break` arrow and enters the code that follows the loop.

This is a flow diagram of the program in Figure 6.31.

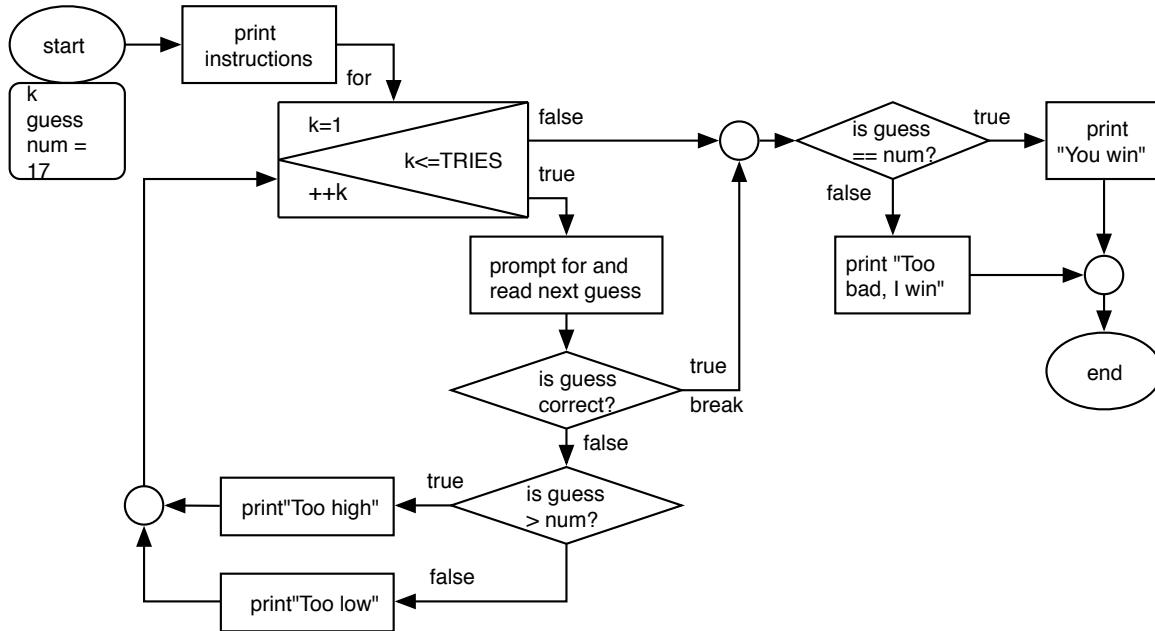


Figure 6.32. A counted sentinel loop.

## 6.5 What You Should Remember

### 6.5.1 Major Concepts

- The `while` loop tests the loop exit condition before executing the loop body. The body, therefore, is executed zero or more times.
- The `while` loop is used for sentinel loops, delay loops, processing data sets of unknown size, and data validation loops when it is important to give the user an error comment different from an ordinary prompt.
- The `for` loop implements the same control pattern as the `while` loop but has a different and more compact syntax.
- The `for` loop is used for counted loops and processing any set of data whose exact size or maximum size is known ahead of time.
- The `do...while` loop executes the loop body before performing the loop exit test. Therefore, the body is always executed one or more times.
- The `do...while` statement is used to form query loops that call a `work()` function repeatedly. It also can be used for data validation.
- A nested loop is used to process the rows and columns of a table.
- A `continue` statement within any kind of loop transfers control back to the top of the loop.
- An `if...break` statement can be used to leave any kind of loop but normally is used to leave a `for` loop before the iteration limit is reached.
- Any one or all the expressions in the loop header of a `for` statement can be omitted. If the loop test is omitted, an `if...break` is used to end the loop. This combination is used for data validation loops.
- The `switch` statement has several clauses, called *cases*, each labeled by a constant. At run time, a single value is compared to each constant and the clause corresponding to the matching constant is executed. A `default` case will be executed if none of the case constants match.

- Most cases in `switch` statements (except the last one) end with a `break` statement. If a case has no `break`, the statements for that case and the next case will be executed.

### 6.5.2 Programming Style

Many programs can be improved by eliminating useless work and simplifying nested logic. The resulting code always is simpler and easier to debug and usually is substantially shorter. Some specific suggestions for improving program style follow:

- The golden rule of style is this: Indent your code consistently and properly.
- Line up the first letter of `for`, `while`, or `switch` with the curly bracket that closes the following block of statements. Indent all the statements within the block.
- Use the `switch` statement instead of a series of nested `if...else` statements when the condition being tested is simple enough.
- Do not compute the same expression twice—compute it once and save the answer in a variable. Any time you do an action twice and expect to get the same answer, you create an opportunity for disaster if the program is modified.
- If two statements are logically related, put them near each other in the program. Examples of this principle are
  1. Initialize a loop variable just before the beginning of the loop.
  2. Do the input just before the conditional that tests it.
- Use the `for` loop effectively, putting all initializations and increment steps in the part of the loop.
- When one control structure is placed within the body of another, we call it *nested logic*. For example, a `switch` statement can be nested inside a loop, and a loop can be nested inside one clause of an `if` statement. Most real applications require the use of nested logic. Some generally accepted guidelines are these:
  1. Keep it simple. An `if` statement nested inside another `if` statement inside a loop has excessive complexity. Many times, the `if` statement can be moved outside the loop or the second `if` statement can be eliminated by using a logical operator.
  2. Establish a regular indenting style and stick to it without exception.
  3. Limit the number of levels of nesting to two or three.
  4. If your application seems to require deeper nesting, break up the nesting by defining a separate function that contains the innermost one or two levels of logic.
- To improve efficiency, move everything possible outside of a loop. For example, when you must compute the average of some numbers, use the loop to count the numbers and add them to a total. Do not do the division within the loop. Moving actions out of the loop frequently shortens the code; it always simplifies it and makes it more efficient. In poorly written programs, “fat loops” account for many of the extra lines. Take advantage of any special properties of the data, such as data that may be sorted, may have a sentinel value on the end, or may be validated by a prior program step and not need validation again when processed.
- During the primary debugging phase of an application, every loop should contain some screen output. This allows the programmer to monitor the program’s progress and detect an infinite loop or a loop that is executed too many or too few times. Debugging output lines should print the loop variable, any input read within the loop, and any totals or counters changed in the loop.

### 6.5.3 Sticky Points and Common Errors

**Semicolons in the wrong places.** The header of a `for` loop or the condition clause of a `while` loop or an `if` statement is, normally, *not* followed by a semicolon.

**Off-by-one errors.** The programmer must take care with every loop to make sure it repeats the correct number of times. A small error can cause one too many or one too few iterations. Every loop should be tested carefully to ensure that it executes the proper number of times. In some counted loops, the loop counter is used as a subscript or as a part of a computation. In these loops, the results may be wrong even when the loop is repeated the correct number of times. This kind of error happens when the value of the loop counter is **off by one** because it is not incremented at the correct time in relation to the expression or expressions that use the counter's value. Consider these two counted loops:

```

for (sumj = j = 0; j < 10; ++j)
    sumj += j;                                // Sum from 0 to 9

// -----
sumk = k = 0;
while (k < 10) {                           // Sum from 1 to 10
    ++k;
    sumk += k;
}

```

These loops are very similar, but **k** is incremented before adding it to **sumk** and **j** is incremented after adding it to the sum. A programmer must decide which timing pattern is correct and be careful to write the correct form.

**Infinite loops.** The **for** loop with no loop test sometimes is called an *infinite loop*, although most such loops contain an **if...break** statement that stops execution under appropriate conditions. Such loops are useful tools. However, a real infinite loop is not useful and should be avoided. Such loops are the result of forgetting to include an update step in the loop body. (In a counted loop, the update step increments the loop variable. In an input-controlled loop, it reads a new value for the loop variable.) During the construction and debugging phases of a program, it is a good idea to put a **puts()** statement in every loop so that you can easily identify an infinite loop when it occurs.

**Nested logic.** When using nested logic, the programmer must know how control will flow into, through, and out of the unit. Statements such as **break** and **continue** affect the flow of control in ways that are simple when single units are considered but become complex when control statements are nested. Be sure you understand how your control statements interact.

#### 6.5.4 Where to Find More Information

- Strings will be introduced in Chapter 12; a sentinel loop that processes a string is shown in Figure 12.14.
- Chapter 19 presents the code for quicksort, one of the best sorting algorithms. Arrays are used with sentinel loops are used in that program.
- Chapter ?? presents two kinds of linked lists; both are typically processed using a sentinel loop.
- Chapter 8, Figure ex-char-work, we show how to use a query loop with character inputs (y/n), which makes a better human interface than numeric responses (1/0).
- Figure 6.32 implements a simple version of a familiar guessing game. We revisit and elaborate on this game in the Random Numbers program on the website.
- The **switch** statement is used with an enumerated type in Figure ???. Figures 12.34 and 15.27 show a very typical use of **switch** to process single-character selections from a menu-interface.
- Appendix D describes the properties and usage of the conditional operator, **? :**, which is the only operator in C that has three operands. The first operand (before the **?**) is a condition. The second operand gives an expression to evaluate if the condition is true. If it is false, the expression after the **:** is evaluated. The result of the conditional operator is the result of whichever expression was evaluated.

#### 6.5.5 New and Revisited Vocabulary

These terms and concepts have been defined or expanded in the chapter:

for loop	loop header	trip count	search loop
loop variable		input validation loop	input-controlled loop
sentinel loop		nested loops	premature exit
sentinel value		delay loop	avoiding <b>break</b>
repeat query		busy wait	constant expression
counted loop		flexible-exit loop	off-by-one error
		infinite <b>for</b> loop	

The following C keywords were discussed in this chapter:

<b>for</b> loop	<b>break</b> statement	<b>switch</b> statement
, (comma operator)	<b>if...break</b> statement	<b>case</b> statement
<b>do...while</b> loop	<b>continue</b> statement	<b>default</b> clause

## 6.6 Exercises

### 6.6.1 Self-Test Exercises

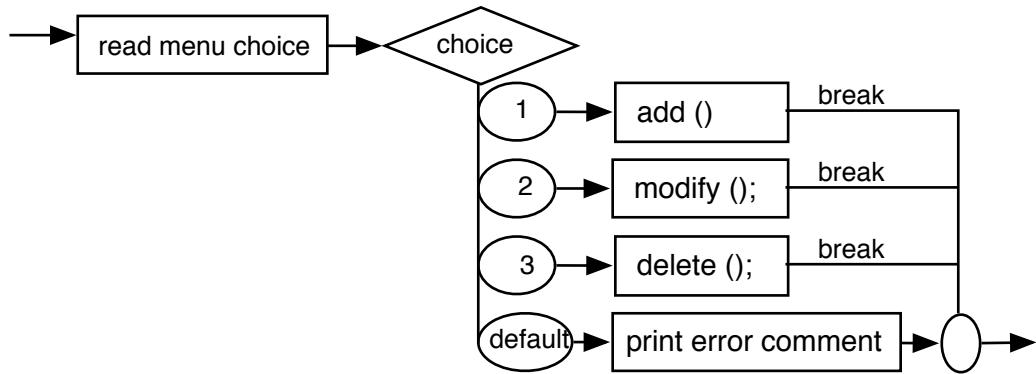
1. The following program contains a loop. What is its output? Rewrite the loop using **for** instead of **while**. Make sure the output does not change.

```
#include <stdio.h>
int main ( void )
{
    int k, sum;
    sum = k = 0;
    while (k < 10) {
        sum += k;
        ++k;
    }
    printf( "A. %i %i \n", k, sum );
}
```

2. The following program contains a loop. What is its output? Rewrite the loop using **while** instead of **do...while**. Make sure the output does not change.

```
#include <stdio.h>
int main ( void )
{
    int k, sum;
    printf( " Please enter an exponent >= 0: " );
    scanf( "%i", &k );
    sum = 1;
    do {
        if (k > 0) sum = 2*sum;
        --k;
    } while (k > 0);
    printf( "B. %i \n", sum );
}
```

3. Explain the fundamental differences between a series of **if...else** statements and a **switch** statement. Under what conditions would you use a **switch** statement? Give an example of a problem for which you could not use a **switch** statement.
4. Explain the fundamental differences between a **while** loop and a **do...while** loop. In what situation would you use each?
5. Given the following fragment of a flow diagram, write the code that corresponds to it and define any necessary variables:



6. Draw a flow diagram for the cash register program in Figure 6.10.
7. Rewrite the following `switch` statement as an `if...else` sequence. (Write code, not a flowchart.)

```

switch (k) {
    case 2:
    case 12: puts( "You lose" ); break;
    case 7:
    case 11: puts( "You win" ); break;
    default: puts( "Try again" );
}
  
```

8. Analyze the following loop and draw a flow diagram for it. Then trace the execution of the loop using the initial values shown. Use a storage diagram to show the succession of values stored in each variable. Finally, show the output, clearly labeled and in one place.

```

for (k = 0, j = 1; j < 3; j++) {
    k += j;
    printf( "\t %i\t %i\n", j, k );
}
printf( "\t %i\t %i\n", j, k );
  
```

9. What does each of the following loops print? They are supposed to print the numbers from 1 to 3. What is wrong with them?

- (a) `for(k = 0; k < 3; ++k) printf( "k = %i", k );`
- (b) `k = 1;
do {
 printf( "k = %i", k );
 k++;
} while (k < 3);`

### 6.6.2 Using Pencil and Paper

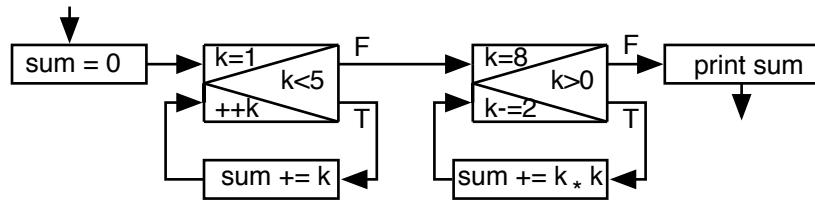
- Explain the fundamental differences between a counted loop and a sentinel loop. What C statement would you use to implement each?
- Explain the fundamental similarity between a `while` loop and a `for` loop. In what situation would you use each?
- Rewrite the following `if...else` sequence as a `switch` statement. Write code, not a flowchart.

```

if (i == 0) puts( "bad" );
else if (i >= 1 && i < 3) puts( "better" );
else if (i == 4 || i == 5) puts( "good" );
else puts( "sorry" );
puts("-----");

```

4. Draw a flow diagram for the cash register program in Figure 6.24.
5. Given the following fragment of a flow diagram, write the code that corresponds to it and define any necessary variables. What number will be printed by the last box?



6. The following program contains a loop. What is its output? Rewrite the loop using `do...while` instead of `for`. Keep the output the same.

```

#include <stdio.h>
int main ( void )
{
    int k, sum;
    for ( sum = k = 0; k < 5; ++k ) sum += k;
    printf( "C. %i %i \n", k, sum );
}

```

7. The following program contains a loop. What is its output? Rewrite the loop without using `break`. Keep the output the same.

```

#include <stdio.h>
int main ( void )
{
    int k, sum;
    for ( sum = k = 1 ; k < 10; k++ ) {
        sum *= k;
        if (sum > 10*k) break;
    }
    printf( "D. %i %i \n", k, sum );
}

```

8. What does the following loop print? It is supposed to print out the numbers from 1 to 3. What is wrong with it?

```

for(k=1; k<=3; ++k);
    printf( "k=%i", k );

```

9. Analyze each loop that follows and draw a flow diagram for it. Then trace execution of the loop using the initial values shown. Use a storage diagram to show the succession of values stored in each variable. Finally, show the output from each exercise, clearly labeled and in one place.

(a)      `k=2;`  
`do { printf("\t %i\n", k); --k; } while (k>=0);`

```
(b)      j=k=0;
        while (j<3){ k+=j; printf("\t %i\t %i\n", j, k); ++j; }
```

10. Given the following code,

```
#include <stdio.h>
int main( void )
{ int j,k;

    for (j = 1; j < 3; j++) {
        for (k = 0; k < 5; k += 2) {
            if (k != 4) printf( " %i, %i ", k, j );
            else k--;
        }
        printf( "\n" );
        if (j % 2 == 1) printf( ">> %i , %i <<\n", j, k );
    }
}
```

- (a) Draw a flowchart that corresponds to the program.
- (b) Using a table like the one that follows, trace the execution of the program. Show the initial values of *j* and *k*. For each trip through a loop, show how the values of *j* and *k* change and what is displayed on the screen. Draw a vertical line between the columns that correspond to each trip through the inner loop.

j	
k	
Output	

### 6.6.3 Using the Computer

1. Sum a series.

Write a program that uses a **for** loop to sum the first 100 terms of the series  $1/x^2$ . Use Figure 6.15 as a guide. Develop a test plan and carry it out. Turn in the source code and output of your program when run on the test data.

2. Sum a function.

Write a function with parameter *x* that will compute the value of  $f(x) = (3 \times x + 1)^{\frac{1}{2}}$ .

Write a main program that sums  $f(x)$  from  $x = 0$  to  $x = 1,000$  in increments of 2. Use a **for** loop. Print the result.

3. Find the best.

Write a program that will allow an instructor to enter a series of exam scores. After the last score, the instructor should enter a negative number as a signal that there is no more input. Print the average of all the scores and the highest score entered.

4. Gas prices.

The example at the end of Chapter 3 (Figure 3.18) is a program that converts a Canadian gas price to an equivalent U.S. gas price. This program does the calculation for only one price. Modify it so that it can convert a series of prices, as follows:

- (a) Remove the price-per-liter input, computation, and output from the main program and put them in a separate function, named **convert()**.

- (b) In place of this code in the main program, substitute a query loop that will allow you to enter a series of Canadian prices. For each, call `convert()` to do the work.
5. An increasing voltage pattern.

Write a program that will calculate and print out a series of voltages.

- (a) Prompt the user for a value of `vmax` and restrict it to the range  $12 \leq vmax \leq 24$ . Let time `t` start at 0 and increase by 1 at each step until the voltage  $v > 95\%$  of `vmax`; `v` is a function of time according to the following formula. Print the time and voltage at each step.

$$v = vmax \times \left(1 - e^{(-0.1 \times t)}\right)$$

- (b) Add a delay loop so that the voltage output is timed more slowly. See Figure 6.19.

6. Sine or cosine?

- (a) Write a double→double function, `f()`, with one `double` parameter, `x`, that will evaluate either  $f_1(x) = x^2 \sin(x) + e^{-x}$  if  $x \leq 1.0$  or  $f_2(x) = x^3 \cos(x) - \log(x)$  if  $x > 1.0$ , and return the result. Write a prototype that can be included at the top of a program.
- (b) Start with the main program in Figure 6.12. Modify the program title and write a new `work()` function that will input a value for `x`, call `f()` using the value of `x`, and output the result.
- (c) Design a test plan for your program.
- (d) Incorporate all the pieces of your program in one file and compile it. Then carry out the test plan to verify the program's correctness.

7. A voltage signal.

An experiment will be carried out repeatedly using an ac voltage signal. The signal is to have one of three values, depending on the time since the beginning of the experiment:

- (a) For time  $t < 1$ ,  $\text{volts}(t) = 0.5 \times \sin(2t)$ .
- (b) For time  $1.0 \leq t \leq 10.0$ ,  $\text{volts}(t) = \sin(t)$ .
- (c) For time  $t > 10.0$ ,  $\text{volts}(t) = \sin(10.0)$ .
- (a) Write a function named `volts()` with `t` as a parameter that will calculate and return the correct voltage. Use the function in Figure 5.18 as a guide.
- (b) Using Figure 6.12 as a guide, write a main program that starts at time 0. Use a loop to call the `volts()` function and output the result. Increment the time by 0.5 after each repetition until the time reaches 12. Print the results in the form of a neat table.

8. Rolling dice.

Over a large number of trials, a “fair” random number generator will return each of the possible values approximately an equal number of times. Therefore, in each set of 60 trials, if values are generated in the range 1...6, there should be about 10 ones and 10 sixes.

- (a) Using the loop in Figure 5.26 as a guide, write a function that will generate 60 random numbers in the range 1...6. Use `if` statements to count the number of times 1 and 6 each turns up. At the end, print out the number of ones and the number of sixes that occurred.
- (b) Following the example in Figure 6.12, write a main program that will call the function from part (a) 10 times. The main program should print table headings; the function will print each set of results under those headings. Look at your results; are the numbers of ones and sixes what you expected? Try it again. Are the results similar? Can you draw any conclusions about the quality of the algorithm for generating random numbers?

9. Prime number testing.

A prime number is an integer that has no factors except itself and 1. The first several prime numbers are 2, 3, 5, 7, 11, 13, 17, and 19. Very large prime numbers have become important in the field of cryptography. The original public-key cryptographic algorithm is based on the fact that there is no fast way to find the prime factors of a 200-digit number that is the product of two 100-digit prime numbers. In this program, you will implement a simple but very slow way to test whether a number is prime.

One method of testing a number  $N$  for primality, is by calculating  $N \% x$ , where  $x$  is equal to every prime number from 2 to  $R = \sqrt{N}$ . If any of these results equals 0, then  $N$  is not a prime. We can stop testing at  $\sqrt{N}$ , since if  $N$  has any factor greater than  $R$ , it also must have a factor less than or equal to  $R$ . Unfortunately, keeping track of a list of prime numbers requires techniques that have not yet been presented. However, a less efficient method is to calculate  $N \% x$  for  $x = 2$  and every odd number between 3 and  $\sqrt{N}$ . Some of these numbers will be primes, most will not. But if any one value divides  $N$  evenly, we know that  $N$  is not a prime.

Write a function that enters an integer  $N$  to test and prints the word **prime** if it is a prime number or **nonprime** otherwise. Write a main program with a query loop to test many numbers.



# **Part III**

# **Basic Data Types**



# Chapter 7

# Using Numeric Types

Two kinds of number representations, integer and floating point, are supported by C. The various **integer types** in C provide exact representations of the mathematical concept of “integer” but can represent values in only a limited range. The **floating-point types** in C are used to represent the mathematical type “real.” They can represent real numbers over a very large range of magnitudes, but each number generally is an approximation, using a limited number of decimal places of precision.

In this chapter, we define and explain the integer and floating-point data types built into C and show how to write their literal forms and I/O formats. We discuss the range of values that can be stored in each type, how to perform reliable arithmetic computations with these values, what happens when a number is converted (or cast) from one type to another, and how to choose the proper data type for a problem.

We would like to think of numbers as integer values, not as patterns of bits in memory. This is possible most of the time when working with C because the language lets us name the numbers and compute with them symbolically. Details such as the length (in bytes) of the number and the arrangement of bits in those bytes can be ignored most of the time. However, inside the computer, the numbers *are* just bit patterns. This becomes evident when conditions such as integer overflow occur and a “correct” formula produces a wrong and meaningless answer. It also is evident when there is a mismatch between a conversion specifier in a format and the data to be written out. This section explains *how* such errors happen so that *when* they happen in your programs, you will understand what occurred.

## 7.1 Integer Types

To accommodate the widest variety of applications and computer hardware, C integers come in two varieties and up to three sizes. We refer to all of these types, collectively, as the *integer types*. However, more than six different names are used for these types, and many of the names can be written in more than one form. In addition, some type names have different meanings on different systems. If this sounds confusing, it is.

The full type name of an integer contains a sign specifier, a length specifier, and the keyword `int`. However,

Common Name	Full Name	Other Acceptable Names	
<code>int</code>	<code>signed int</code>	<code>signed</code>	
<code>longlong</code>	<code>signed longlong int</code>	<code>longlong int</code>	<code>signed longlong</code>
<code>long</code>	<code>signed long int</code>	<code>long int</code>	<code>signed long</code>
<code>short</code>	<code>signed short int</code>	<code>short int</code>	<code>signed short</code>
<code>unsigned</code>	<code>unsigned int</code>		
<code>unsigned longlong</code>	<code>unsigned longlong int</code>		
<code>unsigned long</code>	<code>unsigned long int</code>		
<code>unsigned short</code>	<code>unsigned short int</code>		

Figure 7.1. Names for integer types.

there are shorter versions of the names of all these types. Figure 7.1 lists the commonly used name, then the full name, and finally other variants.

A C programmer needs to know what the basic types are, how to write the names of the types needed, and how to input and output values of these types. He or she must also know which types are portable (this means that the type name always means more or less the same thing) and which types are not (because the meaning depends on the hardware).

### 7.1.1 Signed and Unsigned Integers

In C, integers come in varying lengths and in two underlying varieties: `signed` and `unsigned`. The difference is the interpretation of the leftmost bit in the number's representation. For signed numbers, this bit indicates the sign of the value. For unsigned numbers, it is an ordinary magnitude bit.

Why does C bother with two kinds of integers? FORTRAN, Pascal, and Java have only signed numbers. For most purposes, signed numbers are fine. Some applications, though, seem more natural using unsigned numbers. Examples include applications where the actual pattern of bits is important, negative values are meaningless or will not occur, or one needs the extra positive range of the values.

On paper or in a computer, all the bits in an unsigned number represent part of the number itself. If the number has  $n$  bits, then the leftmost bit has a place value of  $2^{n-1}$ . Not so with a signed number because one bit must be used to represent the sign. The usual way that we represent a signed decimal number on paper is by putting a positive or negative sign in front of a value of a given magnitude. This representation, called *sign and magnitude*, was used in early computers and still is used today for floating-point numbers. However, a different representation, called *two's complement*, is used for signed integers in most modern computers. In the two's complement representation of a signed integer, the leftmost bit position has a place value of -32768. A 1 in this position signifies a negative number. All the rest of the bit positions have positive place values, but the total is negative.

### 7.1.2 Short and long integers.

Integers come in two<sup>1</sup> lengths: `short` and `long`. On most modern machines, short integers occupy 2 bytes of memory and long integers use 4 bytes. The resulting value **representation ranges** are shown in Figure 7.2. As you read this list, keep the following facts in mind:

- The ranges of values shown in the table are the minimum required by the ISO C standard.
- On many 4-byte machines the smallest negative value actually is -32,768 for `short int` and -2,147,483,648 for `long int`.
- Unsigned numbers are explained more fully in Section 15.1.1.
- On many machines in the year 2000, `int` was the same as `short int`. On larger machines, it was the same as `long int`.
- On most machines today, `int` is the same as `long int`.
- The constants `INT_MIN`, `INT_MAX`, and the like are defined in every C implementation in the header file `limits.h`. This header file is required by the C standard, but its contents can be different from one installation to the next. It lists all of the hardware-dependent system parameters that relate to integer data types, including the largest and smallest values of each data type supported by the local system.

The type `int` is tricky. It is defined by the C standard as “not longer than `long` and not shorter than `short`.” The intention is that `int` should be the same as either `long` or `short`, whichever is handled more efficiently by the hardware. Therefore, many C systems on Intel 80x86 machines implement type `int` as `short`.<sup>2</sup> We refer to this as the **2-byte int model**. Larger machines implement `int` as `long`, which we refer to as the **4-byte int model**.

The potential changes in the limits of an `int`, shown in Figure 7.2, can make writing portable code a nightmare for the inexperienced person. Therefore, it might seem a good idea to avoid type `int` altogether and use only `short` and `long`. However, this is impractical, because the integer functions in the C libraries are written to use `int` arguments and return `int` results. The responsible programmer simply must be aware of the situation, make no assumptions if possible, and use `short` and `long` when it is important.

---

<sup>1</sup>Or three lengths, if you count type `char` (discussed in Chapter 8), which actually is a very short integer type.

<sup>2</sup>However, the Gnu C compiler running under the Linux operating system on the same machine implements type `int` as `long`.

Data Type	Names of Constant Limits	Range
<code>int</code>	<code>INT_MIN...INT_MAX</code>	Same as either <code>long</code> or <code>short</code>
<code>short int</code>	<code>SHRT_MIN...SHRT_MAX</code>	$-32,767 \dots 32,767$
<code>long int</code>	<code>LONG_MIN...LONG_MAX</code>	$-2,147,483,647 \dots 2,147,483,647$
<code>longlong int</code>	<code>LLONG_MIN</code>	$9,223,372,036,854,775,807$
<code>longlong int</code>	<code>LLONG_MAX</code>	$-9,223,372,036,854,775,807$
<code>unsigned int</code>	<code>0...UINT_MAX</code>	Same as <code>unsigned long</code> or <code>short</code>
<code>unsigned short</code>	<code>0...USHRT_MAX</code>	$0 \dots 65,535$
<code>unsigned long</code>	<code>0...ULONG_MAX</code>	$0 \dots 4,294,967,295$
<code>unsigned longlong</code>	<code>0...ULLONG_MAX</code>	$0 \dots 18,446,744,073,709,551,615$

Figure 7.2. ISO C integer representations.

**Integer literals.** An **integer literal** constant does not contain a sign or a decimal point. If a number is preceded by a `-` sign or a `+` sign, the sign is interpreted as a unary operator, not as a part of the number. When you write a literal, you may add a type specifier, `L`, `U`, or `UL` on the end to indicate that you need a `long`, an `unsigned`, or an `unsigned long` value, respectively. (This letter is not the same as the conversion specifier of an I/O format.) If you do not include such a type code, the compiler will choose between `int`, `unsigned`, `long`, and `unsigned long`, whichever is the shortest representation that has a range large enough for your number. Figure 7.3 shows examples of various types of integer literals. Note that no commas are allowed in any of the literals.

## 7.2 Floating-Point Types in C

In traditional **scientific notation**, a real number,  $N$ , is represented by a signed **mantissa**,  $m$ , multiplied by a base,  $b$ , raised to some signed **exponent**,  $x$ ; that is,

$$N = \pm m \times b^{\pm x}$$

For example, we might write  $1.4142 \times 10^{-2}$ . A floating-point number is represented similarly inside the computer by a sign bit, a mantissa, and a signed exponent. Figure 7.4 lists the types supported by the standard.

**Floating-point literals.** In traditional mathematical notation, we write real numbers in one of two ways. The simplest notation is a series of digits containing a decimal point, like  $672.01$ . The other notation, called base-10 scientific notation, uses a base-10 exponent in conjunction with a mantissa: we write  $672.01$  as  $6.7201 \times 10^2$ .

In C, real literals also can be written in either decimal or a variant of scientific notation: we write `6.7201E+02` instead of  $6.7201 \times 10^2$ . A numeric literal that contains either a decimal point or an exponent is interpreted as one of the floating-point types; the default type is `double`. (A literal number that has no decimal point and no exponent is an `integer`.) There are several rules for writing literal constants of floating-point types:

---

In this table, we assume that the type `int` is the same length as `long`, and that the maximum representable `int` is `long` is .

Literal	Type	Reason for Type
0	<code>int</code>	Type <code>int</code> is the default.
255U	<code>unsigned</code>	The U code means <code>unsigned</code> .
255L	<code>long</code>	The L code means <code>long</code> .
255UL	<code>unsigned long</code>	You can combine the U and L codes.
32767	<code>int</code>	Largest possible 2-byte <code>int</code>
65536	<code>long</code>	Too large for a 2-byte <code>unsigned int</code>
2147483647	<code>long</code>	Largest possible 4-byte <code>int</code>
3000000000	<code>unsigned long</code>	3 billion, too large for <code>long</code>
6000000000	Compile-time error	6 billion, too large for any integer type

Figure 7.3. Integer literals in base 10.

These are the minimum value ranges for the IEEE floating-point types. The names given in this table are the ones defined by the C standard.

Type Name	Digits of Precision	Name of C Constant	Minimum Value Range Required by IEEE Standard
float	6	$\pm\text{FLT\_MIN} \dots \pm\text{FLT\_MAX}$	$\pm1.175\text{E}-38 \dots \pm3.402\text{E}+38$
double	15	$\pm\text{DBL\_MIN} \dots \pm\text{DBL\_MAX}$	$\pm2.225\text{E}-308 \dots \pm1.797\text{E}+308$
long double	15	$\pm\text{LDBL\_MIN} \dots \pm\text{LDBL\_MAX}$	Same as double currently.

Figure 7.4. IEEE floating-point types.

1. You may write the number in everyday decimal notation: Any number with a decimal point is a floating-point literal. The number may start or end with the decimal point; for example, 1.0, 0.1, .1416, or 120.1.,
2. You may use scientific notation. When you do so, write a mantissa part followed by an exponent part; for example, 4.50E+6. The mantissa part follows the rules for decimal notation, except that it is not necessary to write a decimal point. Examples of legal mantissas are 3.1416, 341.0, .123, and 89. The exponent part has a letter followed by an optional sign and then a number.
  - The letter can be E or e.
  - The sign can be +, -, or it can be omitted (in which case, + is assumed).
  - The exponent number is an integer of one to three digits in the proper ranges, as given in Figure 7.4. If your system does not follow the standard, the ranges may be different.
3. Following the literal value a **floating-point type-specifier** may be used, just as for integers. (This letter is not the same as the conversion specifier of an I/O format.) The specifiers f and F designate a float, while l and L designate a long double. If a floating-point literal has no type specifier, it is a double.

Figure 7.5 shows some examples of floating-point literals and the actual number of bytes used to store them.

### 7.3 Reading and Writing Numbers

Two factors must be considered when choosing a format for reading a number: the type of the variable in which the value will be stored and the way the value appears in the input. Similarly, when printing a number, its type and the desired output format must be considered. In previous examples, we used only a few of the many possible formats for numeric input and output, which we now discuss.

Literal	Type	Size in Common Implementations
3.14	double	8 bytes
1.05792e+05	double	8 bytes
65536E-4f	float	4 bytes
1.01F	float	4 bytes
.021	long double	8, 10, or 12 bytes
171.L	long double	8, 10, or 12 bytes

Figure 7.5. Floating-point literal examples

Context	Conversion	Meaning and Use
scanf()	%d	Read a base-10 (decimal) signed integer (traditional C and ISO C)
	%i	Read a decimal or hexadecimal signed integer (ISO C only)
	%u	Read a decimal unsigned integer (traditional C and ISO C)
	%hi or %hd or %hu	Use a leading h for short int
	%li or %ld or %lu	Use a leading l for long int
printf()	%d	Print a signed integer in base 10
	%i	Same as %d for output
	%u	Print an unsigned integer in base 10
	%hi or %hd or %hu	Use a leading h for short int
	%li or %ld or %lu	Use a leading l for long int

Note: The code for short integers is h instead of s, because s is used for strings (see Chapter 12).

Figure 7.6. Integer conversion specifications.

### 7.3.1 Integer Input

Each type of value requires a different **I/O conversion specifier** in the format string. An integer conversion specifier starts with a % sign, followed by an optional field-width specifier (output only), and a code for the type of value to be read or written. Figure 7.6 summarizes the options available for signed<sup>3</sup> integers. The use of %hi and %li will be illustrated by the program in Figure 7.25.

The %i code is new in ISO C,<sup>4</sup> supplementing the traditional %d. With %i, input numbers can be entered in either decimal or hexadecimal notation (see Chapter 15), whereas %d works only for decimal (base-10) numbers. A representational error<sup>5</sup> will occur if an input value has more digits than the input variable can store. The faulty input will be accepted, but only a portion of it will be stored in the variable. The result is a meaningless number that will look like garbage when it is printed. When a program's output clearly is wrong, it always is a good idea to echo the input on which it was based. Sometimes, this uncovers an inappropriate input format or a variable too short to store the required range of values.

### 7.3.2 Integer Output

For output, the %d and %i conversion codes can be used interchangeably. We use %i in this text because it is more mnemonic for “integer” and therefore less confusing for beginners.

When designing the output of a program, the most important things to consider are that the information be printed correctly and labeled clearly. Sometimes, however, spacing and alignment are important factors in making the output clear and readable. We can control these factors by writing a **field-width** specification (an integer) in the output format between the % and the conversion code (i, li, or hi). For example, %10i means that the output value is an int and the printed form should fill 10 columns, while %4hi means that the output value is a short int and the printed form should fill 4 columns. If the given width is wider than necessary, spaces will be inserted to the left of the printed value. To print the number at the left edge of the field, a minus sign is written between the % and the field width, as in %-10i. The remainder of the field is filled with blanks.<sup>6</sup> If the width is omitted from a conversion specifier or if the given width is too small, C will use as many columns as are required to contain the information and no spaces will be inserted on either end (therefore, the effective default field width is 1). Using a field-width specifier allows us to make neat columns of numbers, as will be illustrated by the program in Figure 7.25.

**Positive or negative?** If an unsigned integer has a bit in the high-order position and we try to print it in a %i format instead of a %u format, the result will have a negative sign and the magnitude may even be small. Unfortunately, most programmers eventually make this careless mistake. The short program in Figure 7.7

<sup>3</sup>Input and output for unsigned numbers will be discussed in Chapter 15, which deals with hexadecimal notation and bit-level computation on unsigned numbers.

<sup>4</sup>Older compilers may not support %i.

<sup>5</sup>Other sources of representational error will be discussed in Section 7.5.

<sup>6</sup>A format string may specify a nonblank character to use as a filler.

---

```
#include <stdio.h>

int main( void )
{
    short unsigned hui = 33000;
    long unsigned lui= 4200000000;

    printf( "%hu is a short unsigned int\n", hui);
    printf( "      printed in hi it is: %hi\n\n", hui );

    printf( "%lu is a long unsigned int\n", lui);
    printf( "      printed in li it is: %hi\n\n", lui );
}
```

---

**Figure 7.7. Incorrect conversions produce garbage output.**

illustrates what can happen when an inappropriate conversion specifier is used. Two unsigned numbers are printed, first properly, then with a signed format.

```
33000 is a short unsigned int
      printed in hi it is: -32536

4200000000 is a long unsigned int
      printed in li it is: -5632
```

In both cases, it is easy to see that the output is garbage because it has a negative sign. However, using a different constant, the output is even more confusing; it is still wrong but there is no negative sign to give us a clue.

```
3000000000 is a long unsigned int
      printed in li it is: 24064
```

### 7.3.3 Floating-Point Input

In a floating-point literal constant (Figure 7.5), a letter (called the **type specifier**) is written on the end to tell the compiler whether to translate the constant as a **float**, a **double**, or a **long double** value. An input format must contain this same information, so that **scanf()** will know how many bytes to use when storing the input value. In a **scanf()** format, the input conversion specifier for type **float** is **%g**; for **double**, it is **%lg**; and for **long double**, it is **%Lg**. Figure 7.8 summarizes the basic conversion specifiers for real numbers. For input, all the basic specifiers **%g**, **%f**, and **%e** have the same meaning and can be used interchangeably, although the current convention is to use **%g**.

The actual input value may be of large or small magnitude, contain a decimal point or not, and be any number of decimal digits long. The number will be converted to a floating-point value using the number of bytes appropriate for the local C translator. (Commonly, this is 4 bytes for **%g**, 8 bytes for **%lg**, and 8 bytes or more for **%Lg**.) However, sometimes, the number stored in the variable is not exactly the same as the input given. This happens whenever the input, when converted to binary floating-point notation, has more digits of precision (possibly infinitely repeating) than the variable can store. In this case, only the most significant digits are retained, giving the closest possible approximation.

### 7.3.4 Floating-Point Output

Output formats for real numbers are more complex than those of integers because they have to control not only the field width but also the form of the output and the number of significant digits printed. There are three basic choices of conversions: **%f**, **%e**, and **%g**. The **%f** conversion prints the value in ordinary decimal form, the **%e** conversion prints it in scientific notation, and the **%g** conversion tries to choose the “best” way to present the number. This may be similar to **%f**, **%e**, or even **%i**, depending on the size of the number relative to the specified field width and precision. Whatever precision is specified and whatever conversion code is used, floating-point numbers will be rounded to the last position printed.<sup>7</sup>

<sup>7</sup>Note that this is different from the rule for converting a real number to an integer, which will be discussed later in this chapter. During type conversion, the number is truncated, not rounded.

Context	Conversion	Meaning and Usage
scanf()	%g, %f, or %e	Read a number and store in a float variable
	%lg, %lf, or %le	Read a number and store in a double variable
	%Lg, %Lf, or %Le	Read a number and store in a long double variable
printf()	%f	Print a float or a double in decimal format
	%e	Print a float or a double in exponential format
	%g	Print a float or a double in general format

Figure 7.8. Basic floating-point conversion specifications.

All three kinds of conversion specifiers (%f, %e, and %g) can include two additional specifications: the total field width (as described for an integer) and a precision specifier. These two numbers are written between the % and the letter, separated by a period, as in %10.3f. In addition, either the total field width or the precision specifier can be used alone, as in %10f or %.3f. The default precision is 6, and the default field width is 1 (as it is for integers). If the field is wider than necessary, the unused portion will be filled with blank spaces.

**The %f and %e conversions.** For the %f and %e conversions, the precision specifier is the number of digits that will be printed after the decimal point. When using the %f conversion, numbers are printed in ordinary decimal notation. For example, %10.3f means a field 10 spaces wide, with a decimal point in the seventh place, followed by three digits. An example is given in Figure 7.9.

In a %e specification, the mantissa is *normalized* so that it has exactly one decimal digit before the decimal point, and the last four or five columns of the output field are occupied by an exponent (an example is given in Figure 7.10). For a specification such as %.3e, one digit is printed before the decimal point and three are printed after it, so a total of four significant digits will be printed.

**The %g conversion tries to be smart.** The result of a %g conversion can look like an integer or the result of either a %f or %e conversion. The precision specifier determines the maximum number of significant digits that will be printed. The printf() function first converts the binary numeric value to decimal form, then it uses the following rules to decide which output format to use. Here, assume that, for a number  $N$ , with  $D$  digits before the decimal point, the precision specifier is  $S$ .

- If  $D == S$ , the value will be rounded to the nearest integer and printed as an integer (an example is

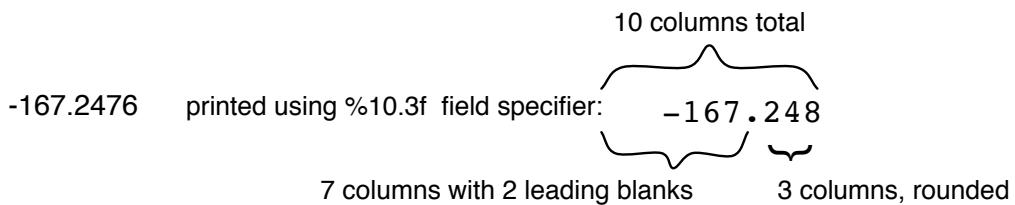


Figure 7.9. The %f output conversion.

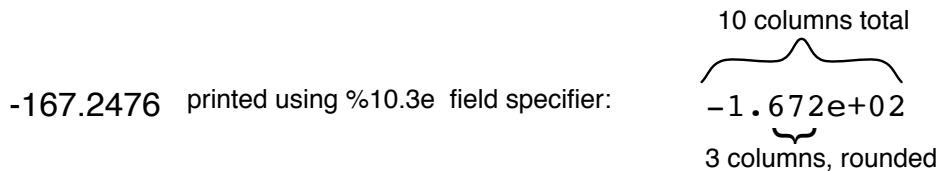


Figure 7.10. The %e output conversion.

-167.2476    printed using %10.3g field specifier:

10 columns total with three digits of precision.

Figure 7.11. Sometimes %g output looks like an integer.

given in Figure 7.11).

- If  $D > S$ , the number will be printed in exponential format, with one digit before the decimal point and  $S - 1$  digits after it (an example is given in Figure 7.12).
- If  $D < S$  and the exponent is less than  $-4$ , the number will be printed in exponential format, with one digit before the decimal point and  $S - 1$  digits after it (an example is given in Figure 7.13).
- If  $D < S$  and the exponent is  $-4$  or greater, the number will be printed in decimal format with  $D$  digits before the decimal point and  $S - D$  digits after it (an example is given in Figure 7.14).

In all four cases, the precision specifier determines the *total* number of significant digits printed, including any nonzero digits before the decimal point. Therefore, %.3g will print one less significant digit than %.3e, which always prints three digits after the decimal point. Also, %.3g may print several digits fewer than %.3f.

Finally, the %g conversion strips off any trailing zeros or decimal point that the other two formats will print. Therefore, the number of places printed after the decimal point is irregular. This leads to an important rule: The %g conversion is not appropriate for printing tables. Usually %f is used for tables, unless the values are of very large magnitude.

### 7.3.5 One Number may Appear in Many Ways

Figure 7.15 shows how the input values of 32.1786594, 2.3, and 12345678 might look if they were read into a float variable, and then printed in a variety of formats. Each input was read using scanf() with the %g

-1672.476    printed using %10.3g field specifier:

10 columns total

3 digits of precision, rounded

Figure 7.12. Sometimes %g looks like %e.

-.000016724    printed using %10.3g field specifier:

10 columns total

3 digits of precision, rounded

Figure 7.13. For tiny numbers, %g looks like %e.

-167.2476	printed using %10g field specifier: (The default precision = 6.)	-167.248
	7 columns with two leading blanks.	3 columns, rounded.

**Figure 7.14.** Sometimes %g looks like %f.

specifier. The actual converted value stored in a `float` variable is shown beneath that. Output of these values was produced by `printf()`, using the conversion formats shown.

**Notes on Figure 7.15. Output conversion specifiers.** When examining the various results, note the following details:

**First line.** This line shows the values entered from the keyboard. In the first column, we input more than the six or seven significant digits that a `float` variable can store; the result is that the last digits of the internal value (on the next line) are only an approximation of the input.

**Second line.** This line shows the actual values stored in three `float` variables. Due to the limited number of bits, the first two values cannot be represented exactly inside the computer. This may not seem surprising for the first value, since a `float` has only six digits of precision, but even the value of 2.3 is not represented exactly. This is because, just as there are repeating fractions in the decimal system (like 1/7), when certain decimal values are converted into their binary representation, the result is a repeating binary fraction. The stored internal value of 2.3 is the result of **truncating** this repeating bit sequence. By chance, even though it is more than six digits long, the third value, 12345678, could be represented exactly. Even though the stated level of precision is six decimal digits, longer numbers sometimes can be represented exactly, while some shorter ones can only be approximated.

**Main portion of table.**

1. All output values are rounded to the last place that is printed.
2. An output too wide for the field is printed anyway, it just overflows its boundary, as in some of the values in the last column.
3. The **default output precision is six decimal places**, so you get six digits after the decimal point with %f and %e unless you ask for more or fewer. With %g, you get a maximum of six significant digits.
4. The %g conversion specifier works similar to %f for numbers that are not too large or too small. The primary differences are that trailing zeros and trailing decimal points will not be printed and that the precision specifies significant digits, not actual digits after the decimal point. For very large and very small numbers, %g works almost like %e except that one fewer significant digit will be printed. Therefore, the number 12345678 printed in %.3e becomes 1.235e+07, but printed in %.3g, it is 1.23e+07.

Input at keyboard	%g	32.1786594	2.3	12345678
Internal bit value		32.17865753173828125	2.2999999523162841796875	12345678
Output using	%f	32.178658	2.300000	12345678.000000
	%e	3.217866e+01	2.300000e+00	1.234568e+07
	%g	32.1787	2.3	1.23457e+07
	%.3f	32.179	2.300	12345678.000
	%.3e	3.218e+01	2.300e+00	1.235e+07
	%.3g	32.2	2.3	1.23e+07
	%10.3f	32.179	2.300	12345678.000
	%-10.3f	32.179	2.300	12345678.000

**Figure 7.15.** Output conversion specifiers.

The programs in Figures 7.18 and 7.25 illustrate some ways in which format specifiers can be used to achieve desired output results. To get a good sense of what the C language does with different floating-point types, experiment with various input values and changing the formats in these programs.

**Alternate output conversion specifiers.** Some compilers will accept `%lg` (or `%lf` or `%le`) in a `printf()` format for type `double`. However, `%g` is correct according to the standard and it is poor style to get in the habit of using nonstandard features. The standard is clear on this issue. All `float` values are converted to type `double` when they are passed to the standard library functions, including `printf()`. By the time `printf()` receives the `float` value, it has become a `double`. So `%g` is used with `printf()` when printing both `double` and `float` values.

However, this is not true for `scanf()`. According to the ISO C standard, you *must* use `%g` for `float`, `%lg` for `double`, and `%Lg` for `long double` in input formats.

## 7.4 Mixing Types in Computations

Since we have introduced both the integer and floating-point data types that are typically used in calculations, it is time to discuss how to use them effectively and convert values from one data type to another.

### 7.4.1 Basic Type Conversions

Two basic types of data conversion can occur, a length conversion and a representation conversion. The **length conversion** occurs between two values of the same data category; that is, between two integers or between two reals. These are **safe conversions** if they lengthen the data representation and thereby do not introduce any representational error. For example, any number that can be represented as a `float` can be represented with exactly the same precision using the `double` type. **Unsafe conversions** may happen if the data representation is shortened.

A **representation conversion** involves switching between two categories, from integer to real or real to integer. Even in systems where a `float` value and an integer value have the same number of bits, their patterns are very different and incompatible. The computer hardware cannot add a `float` to a `long`—one of them must first be converted to the other representation. Depending on the direction of the conversion, it might be classified as safe or not. Therefore, let us examine the basic properties of type conversions more closely.

**Safe conversions.** Converting from a “short” version of a data type to a “long” version is considered to be a safe operation. All the bits stored in the shorter version still can be stored in the longer form, with extra padding in the appropriate positions. Converting from a longer form to a shorter one may or may not be safe. For integers, if the magnitude of the value in the longer form is within the representation range of the shorter one, everything is fine. For real numbers, not only must the magnitude be within the proper range, but the number of significant digits in the mantissa must be small enough as well.

Converting from a `float` to a `double` is safe but the effects can be misleading. The value is lengthened, but it does not increase in precision. A `float` has six or seven decimal places of precision, and the precision of a lengthened value will be the same; the extra bits in the `double` representation will be meaningless zeros. For example, one-tenth is an infinitely repeating fraction in binary. We can store only a finite portion of this value in a `double` and even less in a `float`. Consider the code in Figure 7.16. We initialize both `f` and `d` to 0.1. In both cases, the number actually stored in the variable is only an approximation to 0.1. However, the approximation stored in `d` is more precise. It is accurate up to the 17th place after the decimal point, while `f` is accurate only up to the 8th place. When the value of `f` is converted to type `double` and stored in `x`, it still is accurate only to eight places. The precision does not increase because there is no opportunity to recompute the value and restore the lost bits.

Converting from an integer type to a floating-point type usually is safe, in the sense that most integers can be represented exactly as `floats` and all can be represented exactly as `doubles`. The opposite is not true; most floating-point values cannot be represented exactly as integers.

**Unsafe conversions.** When a value of one type is converted to a shorter type, the number being converted can be too large to fit into the smaller type. We call this condition **representation error**. This is handled quite differently for integers and floating-point numbers.

---

```
#include <stdio.h>
int main( void )
{
    float f = 0.1; // Precision limited to about 7 decimal places.
    double d = 0.1; // Precision limited to about 15 decimal places.
    double x = f; // Converted float; same precision as original value.

    printf( "0.1 as a float = %.17f\n", f );
    printf( "0.1 as a double = %.17f\n", d );
    printf( "0.1 converted from float to double = %.17f\n", x );
}
```

Output:

```
0.1 as a float = 0.1000000149011612
0.1 as a double = 0.1000000000000001
0.1 converted from float to double = 0.1000000149011612
```

---

**Figure 7.16.** Converting a float to a double.

When a large integer is converted to a smaller integer type, only the *least* significant bits are transferred, resulting in garbage. Converting a negative signed number to an unsigned type is logically invalid. Similarly, it is a logical error to convert an unsigned integer to a signed type not long enough to contain the value. The programmer must be careful to avoid any type conversions of this nature, because the C system gives little or no help with detecting the error. Some compilers will display a warning message when potentially unsafe conversions are discovered, others will not. At run time, if such an error happens, no C system will stop and give an error comment.

The shortening action that happens when a `double` value is converted to a `float` has two potential problems. First, it truncates the mantissa, discarding up to nine decimal digits of precision. Second, if the exponent of the value is too large for type `float`, the number cannot be converted at all. In this case, the C standard does not say what will happen; the result is “undefined” and you cannot expect the C system to warn you that this problem has occurred. If it happens in a program, you might observe that some of the output looks like garbage or that certain values are displayed as `Infinity` or `NaN` (not a number).<sup>8</sup>

When a floating-point number is converted to an integer type, the fractional part is lost. To be precise, it is *truncated*, not rounded; the fractional part is discarded, even if it is .999999. To maintain the maximum possible accuracy in calculations, C avoids converting floating-point values to integer types and does so only in four situations, which are listed and explained in the next section. Remember that, in these cases, rounding does not happen, so the floating-point value 1.999999999 will be converted to 1, not to 2.

A last source of unsafe conversions is the use of incorrect conversion specifiers in a `scanf()` statement. For example, using a `%g` (for `float`) in a `scanf()` format when you need `%lg` (for `double`) will not be detected by most compilers as an error. On these systems, the faulty program will compile with no warnings but will not run correctly. It will read the data from the keyboard and convert it into the representation indicated by the format. The corresponding bit pattern, whether the right length or not, will be stored into the waiting memory location without further modification. This will put the wrong information into the variable and inevitably produce garbage results.

## 7.4.2 Type Casts and Coercions

All the different conversions just mentioned can be invoked explicitly by the programmer by writing a type cast. They might also be produced by the compiler because of a type-mismatch in the program code; we call this **type coercion**.

**Type casts.** A **type cast** is an explicit operation that performs a type conversion. A cast is written by enclosing a type name in parentheses and writing this unit before either a variable name or an expression. Any type name can be made into a cast and used as an operator in an expression. Technically, a type cast is a unary operator with precedence lower than all other unary operators but higher than all the binary operators. When applied to an operand, it tells the system to convert the operand to the named type, if possible. Sometimes

---

<sup>8</sup>More is said about these error conditions in Section 7.5.

---

These examples of casts use the following declarations:

```
float x;      double t;      long k;      unsigned v;
```

The starred casts can cause run-time errors that will not be detected by the system and may cause the user's output to be meaningless.

Cast	Nature of Change
* (float) t;	Shortening (possible precision loss and magnitude may be too large)
(double) x;	Lengthening (safe)
(double) t;	No change (this is legal)
(float) k;	Representation conversion (usually safe)
(int) x;	Representation conversion (fractional part is lost)
* (short) k;	Shortening (error if value of k is larger than 32,767)
* (signed) v;	Type relabeling only (error if v > INT_MAX)
* (unsigned long) k;	Type relabeling only (error if k is negative)

Figure 7.17. Kinds of casts.

this adjusts the length of the operand, sometimes it alters the representation, and sometimes it just changes the type labeling. Examples of casts are shown in Figure 7.17.

A cast from a floating-point type to an integer type truncates the value (in the same way that assignment truncates). Casting does not round to the nearest integer; if rounding is needed, it must be done explicitly, by using `rint()` before the value is converted or assigned to an integer variable.

Figure 7.18 contains a simple example of how information can be lost unintentionally during type conver-

---

This program uses type casts and compares the effects of rounding, casting, and assignment.

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

int main( void )
{
    double x, y = 17.7;
    int k, m, n;

    k = (int) y;      // Casting a float to type int truncates
    m = y;           // Assigning a float to an int variable causes truncation.

    printf( "Casting:\t y= %6.2f k= %3i x= %6.2f \n", y, k, x );
    printf( "Assignment:\t y= %6.2f m= %3i x= %6.2f \n", y, m, x );

    n = rint(y);    // Rounding before assignment.

    printf( "Rounding:      y= %6.2f n= %3i \n\n", y, n );

    x = (double) k; // Casting back does not restore the fractional part.

    printf( "Re-casting:   y= %6.2f k= %3i x= %6.2f \n", y, k, x );

    return 0;
}
```

---

Figure 7.18. Rounding and truncation.

---

This program demonstrates the effects of some type coercions. Unlike the program in Figure 7.18, automatic type conversions (not explicit casts) cause the values to change here.

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

int main( void )
{
    float t, w;
    float x = 17.7;
    int k;

    k = x;           // A. The = coerces the float value to type int.
    t = k + 1.0;     // B. The + coerces value of k to type double.
                     //      The = coerces the sum to type float.
    w = sin( x );   // C. Calling sin() coerces x to type double.
                     //      The = coerces the double result of sin() to float.
    printf( "x= %.2f    k= %3i    t= %.2f    w= %.2f\n", x, k, t, w );
}
```

The output is

```
x= 17.70    k= 17    t= 18.00    w= -0.91
```

---

**Figure 7.19. Type coercion.**

sions. We start with a `float` value, convert it into an `int`, and then convert it back again. The values of `y` and `x` are different because information was lost when the value of `y` was cast to `int`. That information cannot be recovered by converting it back again.

#### Notes on Figure 7.18. Rounding and truncation.

**First box: truncation.** In the first line, a cast is used to convert a floating value to an integer value, and the result is stored in an integer variable. This truncates the fractional part of the number, which is lost permanently. The second line has the same effect. The compiler sees that the type of the variable on the left side of the assignment does not match the type of the value on the right, so it automatically generates the (`int`) type cast to make the assignment possible. We say the compiler *coerces* the `double` value to an `int` value. The first two lines of output, below, show the results.

**Second box: rounding.** This box contains a call on `rint()` which rounds `y` to the nearest integer, but returns a value of type `double`. That value is immediately coerced to type `int` and stored in an integer variable. The third line of output, below, shows the result.

**Third box: casting back to type double.** The value of `y` was previously cast to `int` and stored in `k`. Now we take the value of `k` and cast it back to type `double`. Note that this does not (and can not) restore the fractional part; once it is gone, it is gone. The fourth line of output, below, demonstrates these results.

#### *The output.*

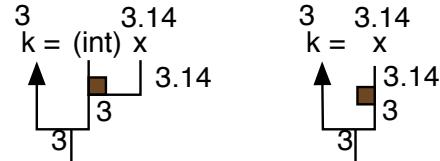
```
Casting:      y= 17.70  k= 17  x= 0.00
Assignment:   y= 17.70  m= 17  x= 0.00
Rounding:    y= 17.70  n= 18
Casting back: y= 17.70  k= 17  x= 17.00
```

**Automatic type coercion.** All of the arithmetic operators defined in Figure 4.1, except `%`, can be used with floating-point types. Within the representational limits of the computer, these operators implement the mathematical operations of addition, subtraction, multiplication, and division. If both operands are `floats`, the result is a `float`. If both are `doubles`, the result is a `double`. If the operands have different types, the compiler will recognize this and attempt to unify the types. A type coercion is a type conversion applied by the compiler to make sense of the types in an expression. C will insert the conversion code before it compiles the operation that requires it. Coercions happen in three basic cases:

1. When the type of the value in a `return` statement does not match the type declared in the function's prototype. If you are performing your calculations carefully, this case should not happen, and many

The examples use these declarations:

```
int k;
double x = 3.14;
```



The diagram on the left is a cast; on the right is a coercion. In both cases, a black conversion box marks the point at which a value is converted from one type to another.

**Figure 7.20. Diagramming a cast and a coercion.**

compilers give warnings when it does occur. It is better style to use an explicit cast in the return statement than to rely on coercion in this case.

2. When the type of an argument to a function does not match the type of the corresponding parameter in the function's prototype. Many of the functions in the math library have `double` parameters and frequently `int` or `float` arguments are passed to them. This is seen in Figure 7.19, line C, where the `float` value of `x` is coerced to type `double`. This kind of coercion may be safe or unsafe. It is normal style to use the safe (lengthening) coercions, and they are used very often. However, coercion should not be used for unsafe (shortening) conversions; use an explicit cast instead.
3. When an arithmetic or comparison operator is used with operands of mismatched types, such as
  - (a) When the value of an expression is being saved into a variable using an assignment statement, as in Figure 7.19, line A. This conversion from the expression type (`float`) to the target type (`int`) is automatic and performed whether safe or not. Examples of coercing a `double` value to type `float` are given in lines B and C. Note that neither of the explicit casts used in Figure 7.18 was necessary. The compiler would have coerced the values into the new formats automatically because a value of one type was being stored in a variable of a different type.
  - (b) When an operator has two real operands or two integer operands, but the operands have different lengths. The shorter value is converted to the type of the longer value, so that no information is lost. Therefore, `short` is converted to `int`, `int` to `long`, and `float` to `double`. The result of the operation will have the longer type.
  - (c) When an operator has operands of mixed representations, as in Figure 7.19, line B. The compiler must convert one value to the type of the other. The rule here is that the conversion always must be done safely, if possible. Therefore, the less inclusive type is converted into the more inclusive type (say, integer to `float` or `double`), so that usually no information is lost. Because of this, most expressions that have a real operand will produce a real result, and many of these are in the `double` format.

**Is coercion a good thing?** Yes, because it allows functions to be easily used with arguments types that are compatible with the parameter types, but not exactly the same. Coercion frees the programmer to think about the calculations, not the representation of the data. Using coercion instead of explicit casts shortens the program and brings the basic calculation into clearer focus. However, it is not wise to depend on coercion unless you are sure that the resulting conversion will be safe, and type warning errors should be eliminated by using explicit casts wherever necessary.

### 7.4.3 Diagramming Conversions

We use parse trees to help us understand the structure of expressions as well as to manually evaluate them. Since coercions and casts affect the results of evaluation, we need a way to show them in a parse tree. Both will be noted on a parse tree as small black squares. Figure 7.20 shows an example of each and Figure 7.21 diagrams casts and coercions within larger expressions.

**Notes on Figure 7.20. Diagramming a cast and a coercion.**

The examples use these declarations:

```
int k=3, r1=1, r2=2;
float w=1.57080;
double x, r_eq;
```

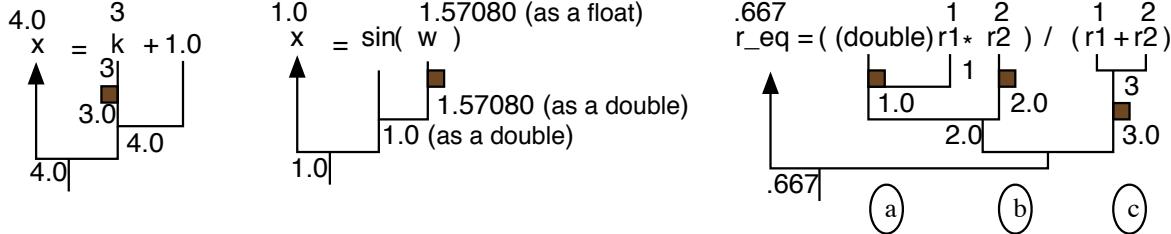


Figure 7.21. Expressions with casts and coercions.

**Diagram on left: a cast.** A type cast is a unary prefix operator; we diagram it with the usual one-armed unary bracket. In addition, we write a black box on the bracket to denote a type conversion. In this example, the real value 3.14 is on the tree above the box; it is converted at the box and becomes the integer 3 below the box.

**Diagram on right: a coercion.** When a real number is stored in an integer variable, it must be coerced first. We represent the coercion by a black box on a branch of the parse tree. Even though no cast is written here, the real value 3.14 above the box is converted at the box to become the integer 3 below the box. The result is the same as if it had been cast.

#### Notes on Figure 7.21. Expressions with casts and coercions.

**Leftmost diagram: coercion of left operand of +.** The first operand of  $+$  is an `int`, the second is a `double` literal. The integer will be coerced to type `double` and real addition will be performed. The result is a `double` stored in `x` with no further conversion.

**Middle diagram: coercion of an argument.** The trigonometric functions in the standard mathematics library are `double`- $\rightarrow$ `double` functions whose arguments must be given in radians. Here we call `sin()` with a `float` argument, which is coerced to `double` before calling `sin()`. The result is a `double` that is stored in `x` with no further conversion.

**Right diagram: a larger expression.** The formula from the third box in Figure 7.23 is diagrammed on the right. In this example, the operand `r1` is explicitly cast (a) from `int` to `double`. This forces the second operand `r2` to be coerced (b) so that real multiplication can happen, producing a `double` value for the numerator of the fraction. The result of the addition in the denominator is an `int`; it is coerced to type `double` (c) to match the numerator. Real division is done and the `double` result is stored in `r_eq`, a `double` variable, with no change.

#### 7.4.4 Using Type Casts to Avoid Integer Division Problems

As discussed earlier, division is an operation whose meaning is quite different for integers and reals; a programmer needs to be aware of these differences. At times, integer division (keeping only the quotient) is a desirable outcome; but at many other times, it is not. Often, even when dividing one integer by another, the fractional part of the answer is needed for the application. Therefore, be careful when writing expressions; divide one integer operand by another only when an integer answer is needed. Otherwise, one of the integers must be cast to a floating-point type before the division. Figures 7.22 and 7.23 illustrate an application in which the use of a cast operation on `int` values achieves the necessary precision in the answer.

**Problem scope:** Find the electrical resistance equivalent,  $r_{eq}$ , for two resistors wired in parallel.

**Input:** Two integer resistance values,  $r_1$  and  $r_2$ .

**Limitations:** The resistances will be between 1 and 1,000 ohms.

**Formula:**

$$r_{eq} = \frac{r_1 * r_2}{r_1 + r_2}$$

**Output required:** The two inputs and their equivalent resistance.

**Computational requirements:** The equivalent resistance must be accurate to two decimal places.

Figure 7.22. Problem specification: computing resistance.

**A division application: Computing resistance.** Figure 7.22 is a simplification of the problem presented in Figure 4.27; it computes the equivalent resistance of two parallel resistors (rather than three). In Figure 7.23, we show how precision can be lost due to careless use of integer division to calculate  $r_{eq}$ . Then we compare this answer to a second value calculated using floating-point variables.

#### Notes on Figure 7.23. Computing resistance.

##### *First box: the input.*

- We use one prompt and one `scanf()` statement to read two input values; the format contains two % codes and we supply two addresses. As long as it is logical and causes no confusion, it is better human engineering to combine the inputs on one line, because this is faster and more convenient for the user.
- We use integer input here because we want to illustrate a potential problem with integer arithmetic. However, the inputs could have been read directly into `double` variables, avoiding the need for the casts or coercions demonstrated next.

We show how to use a type cast or coercion to solve the problem specified in Figure 7.22.

```
#include <stdio.h>

int main( void )
{
    int r1, r2;           // integer input variables for two resistances
    double r1d, r2d;      // double variables for two resistances
    double r_eq;          // equivalent resistance of r1 and r2 in parallel

    printf( "\n Enter integer resistances #1 and #2 (ohms): " );
    scanf( "%i%i", &r1, &r2 );
    printf( "      r1 = %i      r2 = %i \n", r1, r2 );

    r_eq = (r1 * r2) / (r1 + r2);           // Oops! Integer division.
    printf( "      The truncated resistance value is %g\n", r_eq );

    r_eq = ((double)r1 * r2) / (r1 + r2);   // Better: we cast first.
    printf( "      We cast to double first and get %g\n", r_eq );

    r1d = r1;                           // Coerce to type double...
    r2d = r2;                           // by copying into double variables
    r_eq = (r1d * r2d) / (r1d + r2d); // and compute using doubles.
    printf( "      The true value of equivalent resistance is %g\n", r_eq );
}
```

Figure 7.23. Computing resistance.

***Second box: the integer calculation.***

- Since `r1` and `r2` are integers, integer arithmetic will be used throughout the expression and the result will be an integer. The result will be coerced to type `double` after the calculation and before being stored in `r_eq`. Two serious problems arise with this computation, as it is written, that can cause the answer to be less accurate than desired.
- First, the programmer intended to have a real result. You might think that, since the answer is stored in a `double`, it would have a fractional part. But that is not how C works. It does not look at the context surrounding the division to find out what kind of division to perform; it looks only at the two operands, both of which are integer expressions in this case. For two integer operands, it performs integer division, so the fractional part of the result stored in `r_eq` will be 0.
- Second, on a machine with 2-byte `ints`, the result of the multiplication could be a number too large to be represented as an `int`, even when the inputs are relatively small. If this occurs, the overall result will be wrong due to the overflow, a condition we discuss in Section 7.5.

***Third box: using a cast.***

- If any one of the original four operands or the resulting numerator or denominator is cast to a floating-point type, real division will be performed, as demonstrated by the fractional portion of the output.
- Here we cast the first operand of the numerator to `double`, thereby causing real multiplication to be used. Integer addition still will be performed, however, because neither of the operands in the denominator was cast. The result of the addition will be coerced to type `double` before the division is done.

***Fourth box: using double variables and coercion.***

- The output from two runs of this program is shown below. The fractional parts of the correct answers are lost when integer division is used. However, correct answers are obtained when floating-point operations are performed.

```

Enter integer resistances #1 and #2 (ohms): 20 24
r1 = 20    r2 = 24

The truncated resistance value is 10
We cast to double first and get 10.9091
The true value of equivalent resistance is 10.9091

Enter integer resistances #1 and #2 (ohms): 1 2
r1 = 1    r2 = 2

The truncated resistance value is 0
We cast to double first and get 0.666667
The true value of equivalent resistance is 0.666667

```

- When we assign a value to a variable of a different type, the compiler coerces the value to the type of the variable. The integer values entered into the program are transferred from `int` variables into `double` variables. This tells C to find the `double` representation of the numbers `r1` and `r2`. Since integers are a subset of the real numbers, this type conversion is usually safe.

## 7.5 The Trouble with Numbers

Now that we better understand the limitations of the various data types and how conversions between the types occur automatically or at our instruction, we need to consider how to use this knowledge to our advantage. In this section, we discuss how to deal with some computational problems, such as how to properly compare two numbers and what happens when a computed value is outside of the representable range of the data type.

The integer types provide a precise representation for numbers within a restricted range. The restriction is particularly severe for short integers, which are not large enough to store the results of many computations. While the overall range of numbers that can be represented by floating-point types is vastly greater, it still is finite and the representation used is an approximation of the real number with limited precision. We saw some of this precisional error in the last section. The various mathematical operations can produce inaccurate or completely incorrect results if the operands are either too large or too small or if the two operands differ greatly in size. These computational problems are demonstrated in more detail by the following short programs.

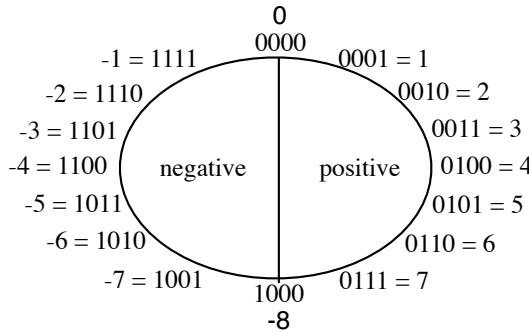


Figure 7.24. Overflow and wrap with a four bit signed integer.

### 7.5.1 Overflow

**Overflow** is the error condition that occurs when the result of an operation becomes larger than the limits of the representation, as described in Figures 7.4 and 7.5. How this error condition is detected and handled differs for the integer and real data types. These overflow situations are a serious problem. They cannot be detected by the compiler. The compiler cannot predict that a result will overflow because it cannot know what data will be used later, at run time, to make calculations. Also, a C system will not detect the error at run time and will not give any warning that it has happened. It usually is possible to look at the results of calculations on the screen and notice when something has gone wrong, but this is far from a desirable solution.

**Integer overflow and wrap.** Integer overflow happens whenever there is a carry *into* the sign bit (leftmost bit) of a signed integer. The result is that the number “wraps around” from positive to negative or negative to positive. **Wrap** is illustrated for 4-bit integers in Figure 7.24. With four bits, we can represent the numbers  $-8\dots+7$ . The four bits represent  $-8, 4, 2, \text{and } 1$  so that, for example, 1101 represents  $-4 + 1 = -3$ .

Suppose we start with the value  $+5$  and repeatedly add 1. We get  $+6$  and  $+7$ , then there is a carry into the leftmost bit (the sign bit), and wrap happens, giving us  $-8$ . If we continue adding 1, we progress, through all possible negative values, toward zero, and back into the positive range of values. Formally, we can say that when  $x$  is the largest positive signed integer that we can represent,  $x + 1$  will be the smallest (farthest from 0) negative integer.

Overflow also happens when an operation produces a result too large to store in the variable that is supposed to receive it. Unfortunately, there is no systematic way to detect overflow or wrap after it happens. Avoidance is the best policy, and that requires a combination of programmer awareness and caution when working with integers. Expressions that cause **integer overflow** are fairly common on small computers because the range of type `int` is so restricted. For a 2-byte signed integer, the largest value is 32,767, which is represented by the bit sequence  $x = 01111111\ 11111111$ . The value of  $x + 1$  is 10000000 00000000 in binary and  $-32768$  in base 10.

Similarly, with unsigned integers, overflow and wrap happen whenever there is a carry *out of* the leftmost bit of the integer. In this case, if  $x$  is the largest unsigned integer,  $x + 1$  will be 0. To be specific, for a 2-byte model, the largest unsigned value is 65,535, which is represented by the bit sequence  $x = 11111111\ 11111111$ . The value, in binary, of  $x + 1$  is 00000000 00000000.

Integer calculations like addition, subtraction, and multiplication with large numbers are likely to exceed the maximum limit, perhaps by quite a lot.<sup>9</sup> On a 16-bit machine, if a computation causes overflow and the result is stored in a variable, only the rightmost (least significant) 16 bits of the overlarge answer will be stored; the rest will be truncated. If the result then is printed, it will appear much smaller than the mathematically correct result (or even negative). Noticing the faulty value is the only way for a user to detect an overflow.

For example, suppose that the 2-byte integer variable `k` contains the number 32300 and you enter a loop that adds 100 to `k` seven times. The value stored in `k` would be, in turn, 32400, 32500, 32600, 32700,  $-32736$ ,  $-32636$ , and finally  $-32536$ . The value has **wrapped** around and become negative, but that does not stop the computer! The program will continue running with a faulty number that is not even approximately correct. For these reasons, 2-byte integers (type `int` on smaller machines and type `short int` on most machines) are not very useful for serious numeric work. We generally use type `long` if we want to use integers in these

<sup>9</sup>Unlike the `*`, `+`, and `-` operators, integer division cannot cause overflow. The smallest integer value you can divide by is 1, which will not increase the magnitude of the value being divided.

calculations. But even though type `long`, with a range up to 2.1 billion, can handle many more calculations properly, it still is limited to 10 digits.

**Floating-point overflow and infinity.** The phenomenon of wrap is unique to integers; floating-point overflow is handled differently. The IEEE floating-point standard defines a special bit pattern, called `Infinity`, that will result if overflow occurs during a computation. (The exponent field of this value is set to all 1 bits, the mantissa to 0 bits.) The constant `HUGE_VAL`, defined in `math.h`, is set to be the “infinity” value on each local system. One way that an overflow can be detected is by comparing a result to `HUGE_VAL` or `-HUGE_VAL`. Systems that implement the full IEEE standard provide the function `finite(x)` in the `math` library, which returns `true` if `x` is a legitimate number or `false` if it is an infinite value or a `NaN` error. Also, in such systems, `printf()` will output overflow values as `+Infinity` or `-Infinity`. Note that the `finite()` function is used to end the `for` loop in Figure 7.25.

**Factorial: A demonstration of overflow.** As an illustration of what all this means in practice, consider the mathematical factorial operation:

$$N! = 1 \times 2 \times \dots \times (N - 2) \times (N - 1) \times N$$

**Factorial**, by its nature, is a function that grows large very rapidly. Figure 7.25 shows a version of the factorial program that computes  $N!$  for values of  $N$  ranging from 1 to 40. The computation is made with variables of five different types so that we can compare the range and precision of these types.

**Wrap error.** The output on the first seven lines is fully correct. (Horizontal spacing has been reduced.)

N	N factorial						
	short	unsigned	int	short int	long int	float	double
1	1	1			1	1	1
2	2	2			2	2	2
3	6	6			6	6	6
4	24	24			24	24	24
5	120	120			120	120	120
6	720	720			720	720	720
7	5040	5040			5040	5040	5040

However,  $7!$  is the largest factorial value that can be stored in a short signed (16-bit) integer. From line 8 on, the numbers in the first column are meaningless; overflow has happened and the answer wraps around to become a negative number. This will not always occur, but when it does, it is a good indication of trouble.

	short	unsigned	int	short int	long int	float	double
8	-25216	40320			40320	40320	40320
9	-30336	35200			362880	362880	362880
10					3628800	3628800	3628800
11					39916800	39916800	39916800

One more value,  $8!$ , can be stored in a short unsigned integer. But, beginning with  $9!$ , the answer overflows into the 17th bit position and the number stored in the variable `factu` is garbage. When working with unsigned numbers, as in the second column, there is not even a negative sign to warn us about the wrap. Nonetheless, the numbers are wrong from line 9 on. This is what makes it so difficult to detect this error in practice. The program suppresses output in these two columns after line 9.

	long int	float	double
11	39916800	39916800	39916800
12	479001600	479001600	479001600
13	1932053504	6227020800	6227020800
14	1278945280	87178289152	87178291200
15	2004310016	1.30767427994e+12	1307674368000

Using long integers, as in the third column, we get correct answers all the way up to  $12!$ , the largest  $N$  for which the calculation can be made using either signed or unsigned long integers. Starting at line 13, the answer for long integers is garbage; it should be the same as the value in the other columns. Here, we get no negative sign to warn us that wrap has occurred, because the value has wrapped past all of the negatives and into the positives again.

---

We compute the factorial function using five different types so that we can compare their range and precision.

```
#include <stdio.h>

int main( void )
{
    int N;                      // Loop counter.
    short int      facts = 1;    // We compute factorial using 5 types.
    short unsigned factu = 1;    // 0! is defined to be 1.
    long int       factl = 1;
    float          factf = 1.0;
    double         factd = 1.0;

    puts( "\n N      N factorial \n      short unsigned \n"
          "      int      short int  long int \t float \t\t\ double \n" );
    // Compute N! using each type, quit after 40 factorial.
    for (N = 1; finite( factd ); ++N) {
        facts *= N;  factu *= N;  factl *= N;
        factf *= N;  factd *= N;

        if (N <= 9)
            printf( "%3i %7hi %7u ", N, facts, factu );
        else
            printf( "%3i ", N );
        if (N <= 17)
            printf( "%12li", factl );
        else
            printf( " " );
        printf( " %18.12g %23.22g\n", factf, factd );
    }
}
```

---

**Figure 7.25. Computing  $N!$ .**

**Representational error.** Between  $N = 14$  and  $N = 34$ , using type `float`, we encounter the limits of the IEEE `float`'s precision, rather than its range. Although we can compute the factorial function for  $N > 13$ , the answers are only approximations of the true answer. (The `float` value computed for  $N = 14$  is 87,178,289,152; this is close to, but smaller than, the true answer, 87,178,291,200, shown in the last column for the `double` calculation.) The `float` simply lacks enough bits to hold all the significant digits, even though the maximum `float` value has not been reached. We say that such an answer is **correct but not precise**. It may be a fully acceptable approximation to the true answer, but it differs in the last few digits. Whether the precision is adequate depends on the application.

	float	double
15	1.30767427994e+12	1307674368000
16	2.0922788479e+13	20922789888000
17	3.55687414628e+14	355687428096000
18	6.40237353042e+15	6402373705728000
19	1.21645096004e+17	121645100408832000
20	2.43290202316e+18	2432902008176640000
21	5.10909408372e+19	51090942171709440000
22	1.12400072481e+21	1124000727777607680000
23	2.58520174446e+22	2.585201673888497821286e+22

Using type `double` (as in the last column) instead of `float` extends the range of accuracy. The same factorial program goes up to  $22!$  with total precision; this number has 18 nonzero digits. For  $N = 23$ , the last few digits show evidence of the error, they should be 664000 not 821286.

---

This program continually divides a number by 10 until the result is too small to store as a normalized `float`.

```
#include <stdio.h>
int main( void )
{
    int N;
    float frac = 1.0;
    puts( "\n Dividing by 10; frac=1/(10 to the Nth power)\n" );
    for (N = 0; N < 50; ++N) {
        printf( " N=%3i    frac= %13.8g    1+frac= %13.8g\n",
                N, frac, 1+frac );
        frac = frac / 10;
    }
}
```

---

**Figure 7.26.** Floating-point underflow.

	float	double
33	8.68331850985e+36	8.683317618811885938716e+36
34	2.95232822997e+38	2.952327990396041195551e+38
35	+Infinity	1.033314796638614422221e+40
36	+Infinity	3.719933267899011774924e+41

At  $N = 35$ , floating-point overflow happens, for the `float` number format where `3.402e+38` is the maximum representable number. However, this does not stop the program, which continues to try to compute the numbers up to  $170!$  before overflow happens using the `double` format. At  $171!$  the test for `finite(facto)` ends the loop.

### 7.5.2 Underflow

The opposite problem of overflow is **underflow**, which occurs when the magnitude of the number falls below the smallest number in the representable range. This cannot occur for integers, only for real numbers, since the minimum magnitude of an `int` is 0. For real numbers, underflow happens when a value is generated that has a **0 exponent<sup>10</sup>** and a **nonzero mantissa**. Such a number is referred to as **denormalized**, which means that all significant bits have been shifted to the right and the number is less than the lowest number specified by the standard. This effect is shown in the left column of the last few lines of output from the division program in Figure 7.26:

N= 43	frac= 9.9492191e-44	1+frac=	1	
N= 44	frac= 9.8090893e-45	1+frac=	1	
N= 45	frac= 1.4012985e-45	1+frac=	1	
N= 46	frac=	0	1+frac=	1
N= 47	frac=	0	1+frac=	1

The program continually divides a value by 10. The actual lower limit of the representation range is `1.175e-38`, and some systems will generate the 0 value when this limit is reached. Others, like the one shown here, still use the denormalized values. But even these, at  $N = 46$ , have all the bits shifted so far to the right that the result becomes 0.

Underflow can result from several kinds of computations:

- Dividing a number by a very large number or repeated division, as just illustrated.
- Multiplying a small number by a near-zero number, which has the same effect as dividing by a very large number.
- Subtracting two values that are near the smallest representable `float` and ought to be equal but are not quite equal because of round-off error.

---

<sup>10</sup>That is, all zero bits in the exponent, which corresponds to a large negative exponent in scientific notation.

### 7.5.3 Orders of Magnitude

The limits of `float` precision can be a problem with addition as well as with multiplication. For example, if you attempt to add a small `float` number to a large one, and their exponents differ by more than  $10^7$  (or 7 **orders of magnitude**), the addition likely will have no effect. The answer will be the same large number that you started with. This is because the floating-point hardware starts the operation by lining up the decimal points of the two operands. In the process, the mantissa bits of the smaller value get denormalized (shifted to the right). But the hardware register in which this happens has a finite width, so the least significant (rightmost) bits of the smaller operand “fall off” the right end of the register and are lost. If the difference in exponents between the operands is great enough, all of the mantissa bits of the smaller value will be lost and only a value of 0 will be left when the addition happens. You can add a millimeter to a kilometer in single precision, but the answer is still 1 kilometer.

This effect is illustrated in the right column of the first few lines of output from the program in Figure 7.26, shown below. This program starts with the value 1.0, divides it repeatedly by 10, and adds each fractional result to 1. After only nine divisions, the original fraction is so small that the addition has no effect. We say that the fraction is insignificant in comparison to 1.0.

```
Dividing by 10; frac=1/(10 to the Nth power)

N= 0   frac=      1   1+frac=      2
N= 1   frac=      0.1  1+frac=      1.1
N= 2   frac=  0.0099999998 1+frac=      1.01
N= 3   frac=  0.0009999993 1+frac=      1.001
N= 4   frac=  9.999999e-05 1+frac=      1.0001
N= 5   frac=  9.999998e-06 1+frac=      1.00001
N= 6   frac=  9.999998e-07 1+frac=      1.000001
N= 7   frac=  9.9999987e-08 1+frac=      1.0000001
N= 8   frac=  9.9999991e-09 1+frac=      1
N= 9   frac=  9.9999986e-10 1+frac=      1
```

**The order of operations.** When dealing with a set of numbers that have highly variable magnitudes, the accuracy of a final result can depend on the order in which operations are performed. For instance, suppose you want the total of a large number of values. If they are all nearly the same size, order does not matter. However, if a few are huge and most are very small, adding up the huge ones first will cause the small ones to be insignificant in proportion to the sum of the large ones. However, if the small ones are added first, their sum may be of an order of magnitude similar to the large values, and therefore, make an important contribution to the overall sum.

Some techniques in numerical analysis also require attention to the magnitude of the numbers that are involved. One example is the Gaussian elimination algorithm for solving a set of simultaneous linear equations. In this algorithm, coefficients of the equations are repeatedly subtracted from, multiplied by, and divided by other coefficients. The subtractions can produce results that are close to, but not quite, zero. But dividing by such a number might cause floating-point overflow. Happily, there is considerable choice about the order in which the coefficients are used, and the solution to this problem is always to process the largest remaining coefficient next.

### 7.5.4 Not a Number

Last, a special value called `NaN`, which stands for “not a number,” can be generated through operations such as `0 / 0`. This is another special bit pattern that does not correspond to a real value. The IEEE standard specifies that any further operation attempted using a `NaN` or `Infinity` as an operand will return the same value. This was seen for `+Infinity` in the factorial example. On our system, the hardware computes `Infinity` and `NaN` values correctly and C’s `stdio` library prints them (as was shown) instead of printing meaningless digits.

Sometimes the order in which a set of calculations is performed can cause an error, while the same operations done in a different order can be correct. For example, suppose we wished to calculate the number of different hands a player might get in the card game Canasta. In this game, a deck has  $n = 104$  cards and each player is dealt a hand of  $k = 11$  cards. The formula for the number of different hands  $H$  can be given two ways:

$$H = \frac{n!}{k! \times (n-k)!} = \frac{n}{k} \times \frac{n-1}{k-1} \times \dots \times \frac{n-k+1}{1}$$

The first formula is the one you are likely to see in a book on probability. However, the number of Canasta hands is calculated using type `float` and using the formula as written, overflow happens. This is shown by

---

Calculate the number of 11-card Canasta hands that can be dealt from a deck of 104 cards. The two calculation methods below are mathematically equivalent, but the first one fails due to overflow. The second one works properly.

**Method 1:** Use the mathematical formula and call the factorial function.

```
float factorial( int n );           // Prototype of factorial function.
combinations = factorial( 104 ) / (factorial( 11 ) * factorial( 104-11 ));
```

**Method 2:** Alternate division and multiplication to keep answer within the range of type `float`.

```
float combinations = 1;    // The answer, so far.
float quotient;          // One term of the formula.
for (int k=11; k>0; --k){
    quotient = (double)(104-k+1) / k;
    combinations *= quotient;
}
```

---

**Figure 7.27.** Calculation order matters.

Method 1 in Figure 7.27, which gives this result:

```
Numerator= inf  Denom1=3.99168e+07  Denom2=inf  Combinations=nan
```

The second method works correctly and gives this answer, which is correct:

```
Combinations= 2.23045e+14
```

### 7.5.5 Representational Error

When two integers are compared, they are either equal or not; this is because we use an exact representation for integers, and they are discrete values (each one differs by exactly 1 from the next). In contrast, the real numbers are not discrete; they are continuous, that is, an infinite number of real values lie between any two we care to write. We can represent some of those numbers exactly but most can be represented only by an approximation. The difference between the true value and its representation is called **representational error**. Types `float` and `double` are **approximate representations** for the real numbers, but with differing precision. As an example, consider this code fragment:

```
float w = 4.4;
double x = 4.4;
printf( " Is x == (double)w? %i \n", (x == (double)w) );
printf( " Is (float)x == w? %i \n", ((float)x == w) );
```

The output, shown below, is unexpected if you forget that the two numbers are represented with limited, and different, **precision** and that the `==` operator tests for exact bit-by-bit equality.

```
Is x == (double)w? 0
Is (float)x == w? 1
```

When the more-precise value is cast to the less-precise type, the extra bits are truncated and the numbers are exactly equal. When the shorter value is cast to the longer type, it is lengthened by adding zero bits at the end of the mantissa, not by recomputing the additional bit values. In general, these zeros are not equal to the meaningful bits in the `double` value.

Computation also can introduce representational error, as shown by the next code fragment. We start with `y`, divide it by a number, then multiply it by the same number. According to mathematics, the result should be the same real number we started with. According to our computer it is, but only sometimes, as with this first set of initial values:

```
float w;
double x, y = 11.0, z = 9.0;

x = z * (y / z);
```

```
w = y - x;
printf( "\n w=%g  x=%.10f \n", w, x );
```

The results from computing this on our system are

```
w=0  x=11.0000000000
```

But if we change the initial values to  $y = 15.0$  and  $z = 11.0$ , the results are different and the value of  $w$  is nonzero:

```
w=1.77635e-15  x=15.0000000000
```

Why does this happen? The answer to a floating-point division has a fractional part that is represented with as much precision as the hardware will allow. However, the precision is not infinite and there is a tiny amount of truncation error after most calculations. Therefore, the answer to  $y / z$  may have error in it, and that error is increased when we multiply by  $z$ . This is why the answer to  $z * (y / z)$  does not always equal the number  $y$  that we started with.

### 7.5.6 Making Meaningful Comparisons

The question then arises, when are two floating-point numbers really equal? The answer is that they should be called *equal* if both are approximations for the same real number, even if one approximation has more precision than the other. Therefore, an approximate test for equality is necessary to compare values that are approximations.

Practical problems often require comparing a calculated value to a specific constant or setpoint or comparing two calculated values that should be equal. Such a comparison is not as simple as it seems, because even simple computations with small floating-point values can have results that differ from the mathematically correct versions. If you read two identical floating-point values into variables of the same floating type and compare them, the values will be equal. However, as soon as you begin to compute, truncation and round-off error can happen. Any computed value could be affected by floating-point representational error. Further, two computed values could be affected by different amounts and in different directions.

Although truncation itself always results in a value smaller than it should be, using a truncated answer as a divisor gives a quotient that is too large. It takes considerable expertise to analyze how severely a number might be affected and in what way. In the example of representational error given previously, doing  $y - (z * (y / z))$  gave a nonzero answer for  $y = 15.0$  and  $z = 11.0$  because of round-off error due to the division, but the same computation on other values of  $y$  and  $z$  gave the answer 0.0. There was no obvious pattern to these zero and nonzero answers when the test was tried with other inputs.

Even though we know that the various results of  $z * (y / z)$  will be very close to the value of  $y$ , the `==` operator tests for exact, not approximate, equality. Since any floating-point value that results from a computation may be imprecise, we cannot use `==` and `!=` on `floats` and `doubles`. We can get around this comparison problem by comparing the *difference* of the two numbers to a preset epsilon value, as in Figure 7.28. We call this an **approximate comparison** for equality with an **epsilon test**. For any given application, we can choose a value of epsilon that is slightly smaller than the smallest measurable difference in the data. We then ask if the absolute value of the difference between the values is less than epsilon—if so, we say the operands are equal. This can be done in one `if` statement by using the absolute value function, `fabs()`, as shown in Figure 7.29.

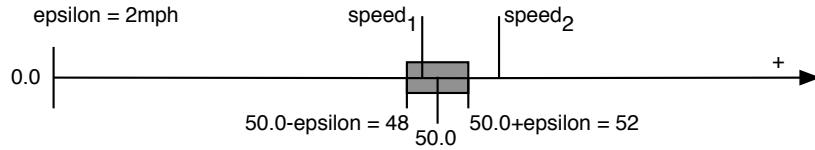
In addition to testing for equality, occasionally we also need to test for a greater-than or less-than condition. In a one-sided test, we still need an epsilon value to compensate for representational error, but the `fabs()` can be omitted. This kind of test is used in Figure 7.31.

### 7.5.7 Application: Cruise Control

Sometimes different actions are required for values below, above, and equal to a target, so we need to use a series of `if` statements to test for these conditions. The program specified in Figure 7.30 and written in Figure 7.31 demonstrates this technique. The figures present an initial version of a cruise control program that would run on a computer embedded in the acceleration system of an automobile to regulate the setting of the automobile throttle. A real cruise control would need to be more complex to avoid drastically overshooting and undershooting the target speed.

**Notes on Figure 7.31. Cruise control.**

With the epsilon shown (2 mph), we say that  $\text{speed}_1 = 49.0$  equals 50.0 because it is within epsilon of 50.0, but  $\text{speed}_2 = 54.0$  does not equal 50.0 with this value of epsilon.



**Figure 7.28. An approximate comparison.**

Use an epsilon test with the absolute value function to compare floating-point values for equality. Note, the `fabs()` function is part of the `math` library and is used with real numbers, as opposed to `abs()`, which is used with integers.

```
double epsilon = 1.0e-3;
double number, target;

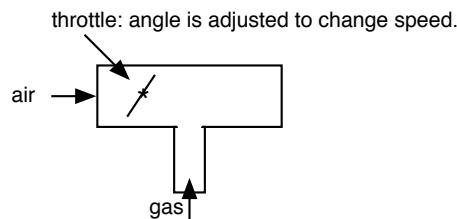
if (fabs( number - target ) < epsilon)    // fabs is floating abs.
    // then we consider that number == target
else
    // we consider the values significantly different.
```

**Figure 7.29. Comparing floats for equality.**

**Problem scope:** A simple version of a cruise control program that would run on a computer embedded in the acceleration system of an automobile.

**Inputs:** These come from the functions `read_on_switch()`, `read_speed()`, `read_throttle()`, and `read_brake()`, which are attached to the car's sensors. Prototypes for these functions are in the file `throttle.h`. There is no direct interaction with a user.

**Formulas:** Increasing or decreasing the angle of the throttle affects the speed. An angle of  $0^\circ$  corresponds to a horizontal throttle and high speed, while the maximum angle of  $90^\circ$  corresponds to a vertical throttle and low speed. At  $90^\circ$ , we assume that some air still can enter the system because the throttle plate is designed to be smaller than the diameter of the tube. We need some air at all times for combustion of the gas-air mixture.



**Constants required:** `eps = 2.5 mph`, the “fuzz factor” for the speed comparison, and `delta = 5°`, the incremental correctional change in throttle setting.

**Output required:** No output report is displayed on a screen. Instead the output is in the form of appropriate calls on the `set_throttle()` function, which controls the position of the car's throttle.

**Figure 7.30. Problem specification: Cruise control.**

***First box: the constants.***

- We define `eps` to be small so that the cruise control system can regulate the speed within a narrow range.
- The throttle setting ranges between  $0.0^\circ$  (horizontal) and  $90.0^\circ$  (vertical); we adjust it by  $5^\circ$  each time we need to raise or lower the car's speed.

***Second box: waiting for the “set” signal.***

- When the cruise control first is turned on, it waits for the driver to press the “set speed” switch. This is performed by the one-line loop that repeats until the “set” signal is received. Note that no actual statement is being executed each time through the loop. The loop simply repeats the test until the test is false. This typically is called a *busy wait loop* and was discussed in Chapter 6.
- As soon as the “set” switch is recognized, we leave the loop and get the input values by calling functions to read the current speed and throttle settings.

***Outer box: responding to conditions.***

The car's throttle setting is increased or decreased in response to the measured speed being too low or too high.

```
#include <stdio.h>
#include "throttle.h"      // Prototypes for switch and throttle functions.

int main( void )
{
    const float eps = 2.5;      // Fuzz factor for comparison.
    const float delta = 5.;     // Change for throttle setting, in degrees.

    float throttle;            // Current throttle setting.
    float target;              // Desired speed setpoint.
    float speed;               // Current speed.
    float dif;                 // Target speed - current speed.

    while (! read_on_switch()); // Leave loop when driver sets speed.
    target = read_speed();     // Initial speed and throttle settings.
    throttle = read_throttle();

    while (! read_brake()){    // Leave loop when driver hits brake.
        speed = read_speed();

        dif = target - speed;   // Compare current speed to target
        if (dif < -eps){
            puts( "Speed is too low; open throttle." );
            throttle -= delta;
            if (throttle > 90.0) throttle = 90.0;
        }
        else if (dif > eps){
            puts( "Speed is too high; close throttle." );
            throttle += delta;
            if (throttle < 0.)  throttle = 0.;
        }
        else puts( "Speed is ok; do nothing." );

        set_throttle( throttle );
    }
}
```

Figure 7.31. Cruise control.

- This loop will monitor the car’s progress. The cruise control will remain active until the driver touches the brake.
- While active, it continually reads the current speed and compares it to the target speed.
- After computing the new throttle setting in the inner box, we generate the output signal of the program by calling the `set_throttle()` function.

***Inner box: adjusting the throttle.***

- Because the speed is a `float`, we use an epsilon test; that is, we declare the numbers to be equal if they differ by less than epsilon (2.5 mph).
- Here it is not just important to know whether the speeds are the same but, when they are different, which is greater. We use an `if...else` sequence to handle this.
- If the current speed is slower than the target speed minus epsilon, we subtract delta from the throttle setting. If the current speed is too fast, we add delta. (An actual cruise control algorithm would use more information than just the current speed to make this decision.) If both these tests fail, then the speeds would be “equal” according to our original approximate-equality test.

## 7.6 What You Should Remember

### 7.6.1 Major Concepts

***Integers and their properties.***

- *Integers come in many forms in C: `signed` and `unsigned`, `short` and `long`.* An integer type permits us to represent a limited range of values exactly. The `short signed` integers can store numbers only up to 32,767. The largest `long signed` integer is 2,147,483,647. The largest `long unsigned` integer is 4,294,967,295. If numbers larger than this are needed, a floating-point type must be used.
- *Literal integers.* A literal integer is a sign followed by a sequence of digits. The exact type of a literal depends on its sign, its magnitude, and the range of type `int` on the supporting computer hardware. The digits may be followed by a letter L or U to indicate that the number should be long or unsigned. C supports literals in bases 8 (octal), 10 (decimal), and 16 (hexadecimal). A literal integer is interpreted as octal if it starts with a 0 digit or hexadecimal if it starts with 0x. Otherwise it is a decimal number.
- *Integer input formats.* Integers can be read using `%i` or `%d`. The `%i` is more general and can input base ten and hexadecimal numbers. The `%d` is older and is limited to base ten (decimal) inputs. Long and short integers require different format specifiers: `%li` or `%ld` for long and `%hi` or `%hd` for short. When reading, it is important to use the format specifier that matches the type of the variable that will receive the input. Many compilers do not check for correct specifiers, and using an incompatible specifier will cause strange and unpredictable results.
- *Integer output formats.* Integers can be printed using `%i` or `%d`, with an optional field width specification between the percent sign and the letter. As with input, long and short integers require different format specifiers: `%li` or `%ld` for long and `%hi` or `%hd` for short. When printing a value it is important to use the format specifier that matches its type.
- *When making calculations with large integers,* the programmer must be wary of integer overflow and wrap. If a variable contains the maximum integer value, adding 1 will cause the value to wrap. The answer will be the minimum representable negative value (farthest from zero).

***Floating-point numbers.***

- *C supports floating-point numbers in two or three lengths.* These types, named `float`, `double`, and `long double`, are used to represent real numbers. As with scientific notation, a floating-point number has an exponent that encodes the order of magnitude of the number and a mantissa that encodes the numeric value to a limited number of places of precision. The limits of floating-point representation were examined in Figure 7.4.

- *The type `double` is the most important* of the three floating-point types, because the C mathematics library is written to process `double` numbers (not `float` or `long double`) and does all its computations with `doubles`. Type `float` exists to give the programmer a choice; `double` provides twice as much precision and a much larger range of exponents but takes twice as much storage space as `float` and may take twice as long to process in a computation. When memory space and processing time do not matter, many programmers use `double` because it provides more precision.
- *Type `float` vs. `double`.* Because a programmer can combine types `float` and `double` freely in expressions, most of the time, it does not matter which real type is used. Sometimes the degree of precision required for the data dictates the use of `double`. Since all the functions in the math library expect `double` arguments and return `double` results, some programmers just find it easier to declare all real variables as `double`.
- *Floating-point literals.* A floating-point literal may be written with a decimal point or in scientific notation. In the first case, the decimal point may be written before the first digit, after the last digit, or anywhere in between. A literal in scientific notation starts the same way, but then has an exponent part: the letter `E` or `e`, an optional + or - sign, and an integer exponent in the range 0...38 for type `float` or 0...308 for type `double`.
- *Floating-point input formats.* For input, we have been using the type specifiers `%g` for `float` and `%lg` for `double`. The specifiers `%f` and `%lf` are also appropriate and commonly used. It is essential to use the format specifier that matches type of the variable in which the input will be stored. Otherwise, the results will be wrong and appear to be garbage.
- *Floating-point output formats.* For output, there are three formatting strategies, with a different specifier for each. The basic type specifier is `%g` for both `float` and `double`. It will print the output in whatever format seems most appropriate for the size of the number. (No letter 1 is needed or even permitted by the standard, although many compilers will accept `%lg` and do the right thing.) Optional width and precision specifications may be written between the percent and the `g`.

The specifiers `%e` and `%f` are used when the programmer needs more control over the appearance or position of the output number. When `%e` is used, the output will be printed using scientific notation and `%f` is used with width and precision specifications to print numbers in neat columns.

- *Floating-point computations* can also cause overflow. This happens when you divide by a near-zero value or multiply two very large numbers. The number will have an exponent of all 1-bits. On some systems this will be printed as `+Infinity`; on others, it will be a number with a very large exponent.
- *Underflow.* This error condition happens when a real number becomes very close to zero but does not exactly equal zero. In this situation, some systems store the number in a denormalized form, others simply set the result to 0.
- *Comparing reals.* Computations on real numbers commonly introduce small and somewhat unpredictable representational errors. For this reason, all comparisons between computed reals should be made using a tolerance.

### *Choosing the proper data type.*

- *An integer type or a real type?* For most problems the details of the specification will make it rather obvious whether an integer or real data type should be used to represent a particular entity. Integers typically are used for such things as loop counters, simple quantities, menu choices, and answers to simple questions. Real variables more typically are used for measurements and mathematical calculations.
- *Two other important issues.* Memory limitations and speed of execution must sometimes be considered in choosing a data type. If you are processing large amounts of data and precision is not important, then `float` variables use only half as much space as `doubles` and an `int` may use even less (depending on the compiler and computer system). If speed is of concern, integer arithmetic is performed more quickly than real computations on many machines. So if integers can be used, do so. Otherwise, in general, computations involving `float` values are faster than those using `doubles`, due to the smaller amount of information (number of bits) being processed.

### *Computational issues.*

Group	Operators	Complications
Casts	(int)	Conversion from <code>double</code> or <code>float</code> will discard the fractional part.
	(short)	Conversion from a <code>long</code> will produce a garbage result if the value of the <code>long</code> is too great to fit into a <code>short</code> .
	(float)	Conversion from <code>int</code> is safe; from <code>double</code> , precision may be lost.
Coercions	=	Loss of precision does occur during assignment of a more-precise value to a less-precise variable.
	parameters	Argument values are coerced to match the declared types of the parameters.
	return values	The value returned by a function is coerced to match the declared function return type.

Figure 7.32. Casts and conversions in C.

- An integer in the computer is an exact representation of the corresponding mathematical integer. However, each size of computer integer has a limited range and cannot store a number outside that range. An attempt to do so causes overflow and wrap.
- A floating-point number is an approximate representation of the corresponding mathematical real number. Computations with type `float` and `double` are subject to possible overflow, underflow, and loss of precision. After overflow or underflow, further computation is meaningless and is trapped by some (but not all) contemporary C systems. Such numbers are labeled `NaN`.

#### *Casts and mixed-type operations.*

- C supports mixed-type arithmetic. Integer and floating-point types can be mixed freely in arithmetic expressions. When two values of differing types are used with an operator, the value with less precision automatically is coerced to the more precise representation.
- If an integer is combined with a `float` or a `double` in an expression, the integer operand always is converted to the type of the floating-point operand before the operation is performed; the result of the operation is a floating-point value.
- A type conversion may be “safe,” in that it will cause no loss of information, or it may be “unsafe,” because it can cause a loss of precision or simply result in total garbage. Knowing when a type conversion can be used safely is important. However, sometimes an unsafe conversion is exactly what the programmer needs.
- An explicit type cast must be used to perform real division with integer operands.

### 7.6.2 Sticky Points and Common Errors

**Operators.** The table in Figure 7.32 gives a brief summary of the difficulties that might be encountered when using C casts and conversions.

**Algorithms.** Know the weak points in your algorithm as well as any assumptions on which the calculations might be based. If the algorithm can “blow up” at any point, guard against that possibility.

**Formats.** Using the wrong conversion specifier in a format can cause input or output to appear as garbage. Default length, `short`, and `long` integers have different conversion codes, as do `signed` and `unsigned` integers.

**Debugging.** Insert printouts into your program after every few calculations to spot potential calculation errors.

**Precision.** When using reals, there is no way to tell from the printed output whether a value came from a `double` or a `float` variable. If you specify a format such as `%.10f`, you might see 10 columns of nonzero digits printed, but that does not mean that all 10 are accurate. If the number came from a `float` variable, the eighth through tenth digits usually will be garbage. A similar problem happens when the precision specification of the output is made greater than the actual precision of the input. If an answer was calculated from input having two places of precision, all decimal positions in the output after the second will be meaningless. Remember that it is up to you to limit the columns of output to the precision of the number inside the machine or the known accuracy of the calculation, whichever is smaller.

### Avoiding and handling runtime errors.

- Do not try to add or subtract values of widely differing magnitudes.
- If there is any possibility that a divisor could be zero, test for it!
- Define an epsilon value, related to the precision of the input, which is the smallest meaningful value in this context. Any number whose absolute value is smaller than epsilon should be considered zero and any two numbers whose difference is less than epsilon can be considered equal.
- An output with a huge and unreasonable exponent means it is probably the result of an overflow. Each programmer needs to be able to recognize the overflow and undefined values that will be printed by the local compiler and system. If these values appear in the output, the programmer should identify and correct the erroneous computation that caused them.
- Not a number. The C standard does not specifically cover how a compiler must handle the special values `NaN`, `+Infinity`, and `-Infinity`; it leaves these results officially “undefined”. This means that a particular compiler may do anything that is convenient about the problem. Many do nothing; a garbage result is returned and the user is not notified that there is an error. However, most computer hardware will set an error indicator when the various floating-point problems occur. This permits a program to test for a particular result and thereby discover these illegal operations. The user can get control by defining a signal handler<sup>11</sup> to trap these types of signals and process them. However, most programmers have no idea how to do this, and most user programs don’t attempt to use the interrupt system. Avoidance is the best policy for the ordinary programmer. The careful programmer takes these precautions:

#### 7.6.3 Programming Style

- It is appropriate to use integers for loop counters and customary to give them short names such as `j`, `k`, `m`, and `n`.
- Although the letters `i` and `l` have traditionally been used to name integer counters, they are poor choices, because they are easily mistaken for each other and for the numeral 1.
- Floating-point numbers traditionally have been given names starting with `f...h` and `r...z`.
- When implementing standard scientific or engineering formulas, it makes sense to use whatever variable names are used traditionally to express that formula, even when those names are single letters. Otherwise, use variable names long enough to convey the meaning or purpose of the variable.
- Use a `%f` conversion specifier if your output needs to be in neat columns. Use `%g` if you have no good idea whether the value to be printed is large or small. Use `%e` if the range of values is extreme.
- To print a table in neatly aligned columns, use a `%f` conversion specifier and include a field width. The `%g` conversion is not appropriate for tables.

### Portability.

- There is some variation among compilers in the way floating-point types are handled. Sometimes the underlying computer hardware does not support floating-point arithmetic, in which case floating-point representation and computation must be emulated by software. Emulation, of course, is much slower.
- Although all ISO C compilers must permit use of the type name `long double`, many simply make it a synonym for `double`.

---

<sup>11</sup>This subject is beyond the scope of this text.

- Some systems use 2 bytes to represent an `int`, others use 4 bytes. The two lengths of integers make portability of code a nightmare. Unless a programmer is aware of the different meanings of `int` and assiduously avoids relying on the size of his `ints`, it is very unlikely that his or her programs will run on each kind of machine without additional debugging. Furthermore, errors due to integer sizes are among the hardest to find because of the ever-present automatic size conversions all C translators perform.

It would be nice to avoid type `int` altogether and use only `short` and `long`. However, this is impractical because the integer functions in the C library are written to use `int` arguments and return `int` results. So what should the responsible programmer do?

1. Be aware.
2. Use `short` or `long` when the length is important in your application.
3. Do not rely on assumptions about the size of things.
4. Check all the possible data coercions and conversions and think about what can be done for those labeled *unsafe*.

**Algorithms.** Know the weak points in your algorithm as well as any assumptions on which the calculations might be based. If the algorithm can “blow up” at any point, guard against that possibility. Do not try to add or subtract values of widely differing magnitudes.

**Debugging.** Insert printouts into your program after every few calculations to spot potential calculation errors.

**Handling error conditions.** The C standard does not specifically cover how a compiler must handle the special values `NaN`, `+Infinity`, and `-Infinity`; it leaves these results officially “undefined.” This means that a particular compiler may do anything convenient about the problem. Many do nothing; a garbage result is returned and the user is not notified of an error. However, most computer hardware will set an error indicator when the various floating-point problems occur. This permits a program to test for a particular result and thereby discover the illegal operations. The user can get control by defining a signal handler<sup>12</sup> to trap these types of signals and process them. However, most programmers have no idea how to do this, and most user programs don’t attempt to use the interrupt system. Avoidance is the best policy for the ordinary programmer. The careful programmer takes these precautions:

- An output with a huge, unreasonable exponent probably is the result of an overflow. Each programmer needs to be able to recognize the overflow and undefined values that will be printed by the local compiler and system. If these values appear in the output, the programmer should identify and correct the erroneous computation that caused them.
- If a divisor possibly could be 0, test for it.
- Define an epsilon value, related to the precision of the input, that is the smallest meaningful value in this context. Any number whose absolute value is smaller than epsilon should be considered 0 and any two numbers whose difference is less than epsilon can be considered equal.

#### 7.6.4 New and Revisited Vocabulary

These are the most important terms and concepts from this chapter that deal with the representation of numbers:

representation range	IEEE floating-point standard	precision
integer overflow	exponent	normalized and denormalized
wrap	mantissa	order of magnitude
scientific notation	floating-point overflow	correct but not precise
approximate representation	underflow	order of performing operations
	representational error	

These are the most important terms and concepts from this chapter that deal with the C language:

<sup>12</sup>This subject is beyond the scope of this text.

integer types	floating-point types	type coercion
2- and 4-byte models	floating-point type specifier	I/O conversion specifier
integer type specifier	floating-point literal	field width specifier
integer literal	representation conversion	precision specifier
literal modifiers	type cast	default output precision

These are the most important terms and concepts from this chapter that deal with numeric algorithms:

integer division	indeterminate results	approximate comparison
division by 0	safe conversions	epsilon test
truncation	unsafe conversions	factorial
rounding	length conversion	Euclid's method for square root

The following conversion specifiers, functions, prototypes, and library files were discussed in this chapter:

const	double	+Infinity and -Infinity
int	long double	finite()
i and d conversions	e and le conversions	HUGE_VAL
long int	f and lf conversions	NaN
li and ld conversions	g and lg conversions	limits.h
short int	INT_MIN	float.h
hi and hd conversions	INT_MAX	rint()
signed int	FLT_MIN	sin()
unsigned int	FLT_MAX	cos()
u, hu and lu conversions	DBL_MIN	abs()
float	DBL_MAX	fabs()

### 7.6.5 Where to Find More Information

- Unsigned integer types and the bitwise operators that work on them are covered in Chapter 15.
- The last primitive data type, pointers, is introduced in Chapter 11 (pointer parameters), and discussed extensively in Chapter 16 (dynamic allocation), Chapter 16 (pointer algorithms), Chapter ?? (linked lists), and Chapter 20 (pointers to functions).
- The IEEE Standard for floating point computation can be found through this website  
[en.wikipedia.org/wiki/IEEE\\_Floating\\_Point\\_Standard](https://en.wikipedia.org/wiki/IEEE_Floating_Point_Standard)
- A list of disasters caused by numeric errors can be found on the web by searching for "space program disaster overflow". Among the incidents listed there are:
  - Failed Navy rocket launches, 1999: bad decimal point.
  - Ariane explosion, 1996: Large float converted to integer, causing overflow.
  - Patriot-Scud, 1991: rounding error.
  - Loss of Mars orbiters, 1999: mixture of pounds and kilograms.
  - USS Yorktown "dead in the water", 1998: input and division by 0.

## 7.7 Exercises

### 7.7.1 Self-Test Exercises

- The following functions and constants are all defined in the standard ISO C libraries. Name the specific header file that must be #included to use each one.
  - HUGE\_VAL
  - sin() and cos()
  - INT\_MAX
  - finite()

- (e) `scanf()`  
 (f) `fabs()`  
 (g) `rint()`  
 (h) `FLOAT_MAX`
2. What is the type of each of the following integer literals in a C compiler, where type `int` is the same length as type `short`? If the item is not a legal literal, say so.
- (a) 33333  
 (b) 10U  
 (c) 32270  
 (d) -20  
 (e) 3000000000  
 (f) 100L  
 (g) 32,767  
 (h) 65432
3. Will the result of each of the following expressions be true or false? All variables are type `int`. Use the integer data values `k = 3`, `m = 9`, and `n = 5`.
- (a) `m == k * 3`  
 (b) `k * (9 / k) == 9`  
 (c) `k * (n / k) == n`  
 (d) `k = n`
4. What will be stored in `k` or `f` by the following sets of assignments? Use these variables: `int h, k, m;` `float f; double g;`.
- (a) `f=1.6; k = f;`  
 (b) `f=1.4; k = (int) f;`  
 (c) `g=5.1; f = (float) g;`  
 (d) `g=9.6; k = (float) g;`  
 (e) `g=9.7; k = g + 1.8;`  
 (f) `h=13; m=4; f = (float) h / m;`  
 (g) `h=13; m=4; f = (float)(h / m);`  
 (h) `g=1.02; f = 10.2 f == g * 10;`
5. Draw a parse tree for each of the following computations (include conversion boxes). Then use the given data values to evaluate each expression and record the final values stored in the variables `f`, `g`, and `k`. Use these declarations and initial values: `int k, j=70; float f=32.08; double g=10.0;`.
- (a) `f = g * (int) f + j;`  
 (b) `k = g * (int) f + j;`  
 (c) `g = pow( f, 10.0);`  
 (d) `g = pow( f, f);`
6. Draw a parse tree for each of the following computation (include conversion boxes). Then use the given data values to evaluate each expression and record the final values stored in the variables `J`, `L`, and `F`. Indicate if overflow occurs during an evaluation.

```
short int J, K=100;
long int L, M=2000;
float F;
```

- (a) `J = L = K * K * K;`

- (b) `L = M * M * M;`  
 (c) `F = M * M * M;`
7. We can represent all integer values using the `double` representation. List two situations in which we would still want to use the `int` data type.
8. Given the variable declaration `double x = 1234.5678;`, what is printed by the following statements?
- `printf( "%e %f %g", x, x, x );`
  - `printf( "%10.3e %10.3f %10.3g %10.5g", x, x, x, x );`
9. Given the following variable declarations and input prompt, what is stored in `k`, `m`, `x`, or `d` by the following statements when the user enters the number shown on the left of each item? (If the result is garbage, say so.)
- ```
short int k;
long int m;
float x;
double d;
printf( " Please enter a number: " );
```
- |                  |                                      |
|------------------|--------------------------------------|
| (a) 33           | <code>scanf( "%hi", &amp;k );</code> |
| (b) 33000        | <code>scanf( "%hi", &amp;k );</code> |
| (c) -44000       | <code>scanf( "%li", &amp;k );</code> |
| (d) 33           | <code>scanf( "%li", &amp;m );</code> |
| (e) 33           | <code>scanf( "%g", &amp;d );</code>  |
| (f) 109e-02      | <code>scanf( "%lg", &amp;d );</code> |
| (g) 123.456789   | <code>scanf( "%lg", &amp;x );</code> |
| (h) -43.21098765 | <code>scanf( "%f", &amp;x );</code>  |
10. Answer the following questions about the computer you use. Write a short program to find the answers, if necessary.
- What is the largest `int` that you can enter on your machine and print correctly?
  - What is the biggest `unsigned int` you can read and write?
  - When is  $x + 1 < x$ ?
11. Write one or a few lines of code that will cause integer overflow and wrap to happen.
12. Say whether each of the following computations will give a meaningful answer or is likely to cause overflow, underflow, or a serious precision error.
- ```
float f;
float g = 0.1
float h = cos(0); // This should be 1.0
```
- |  |
|--|
| (a) <code>f = 0.000001 - pow(g, 5);</code> |
| (b) <code>f = 233344455.5 * .1;</code>     |
| (c) <code>f = 233344455.5 + .1;</code>     |
| (d) <code>f = pow(3.14159, 100);</code>    |
13. When a floating-point number is printed in `%e` format, it is printed in normalized form, with exactly one digit to the left of the decimal point. Rewrite the following numbers in normalized scientific notation:
- 75.23
  - .00012

- (c) .9998
- (d) 32,767

14. Each item that follows compares two numbers. For each, answer whether the result is `true`, `false`, or indeterminate and explain why. To get the correct answers, you must know about the type conversions used in mixed-type expressions.

```
float w = 3.3;
int j = w, k = 3;
double x = 3.0, y = 3.3, z = 4.2;
```

- (a) `x == k`
- (b) `y == k`
- (c) `x != y`
- (d) `w == j`
- (e) `w == y`
- (f) `x == w`
- (g) `(float)x == w`
- (h) `y == z * (y / z)`
- (i) `x + 1.0 == k + 1`
- (j) `x == .3 * 10`

### 7.7.2 Pencil and Paper

1. Draw a parse tree for the following computation (include conversion boxes). Then use the tree to evaluate the expression. Use these variable declarations and initial values: `int k, j=10; double g=402.5; float f=32.08;`.

```
k = g - (int) f * j;
```

2. Given the variable declarations, what is printed by the following statements?

```
int k = 1234;
float x = 1681.700612;
float y = 23.28765;
```

- (a) `printf( "k =%i\n", k );`
- (b) `printf( "k =%10i\n", k );`
- (c) `printf( "k =%-10i\n", k );`
- (d) `printf( "x = %10.3f \n", x );`
- (e) `printf( "x = %10.4f \n", x );`
- (f) `printf( "x = %10.4e\n", x );`
- (g) `printf( "x = %.3g\n", x );`
- (h) `printf( "y = %.3g\n", y );`

3. Given the following variable declarations and input prompt, what is stored in `m`, `x`, or `d` by the statements when the user enters the number shown on the left of each item? (If the result is garbage, say so.)

```
long int m;
float x;
double d;
printf( " Please enter a number: " );
```

- |            |                                      |
|------------|--------------------------------------|
| (a) 33000  | <code>scanf( "%li", &amp;m );</code> |
| (b) -44000 | <code>scanf( "%hi", &amp;m );</code> |

- |                    |                                      |
|--------------------|--------------------------------------|
| (c) 76.5           | <code>scanf( "%g", &amp;x );</code>  |
| (d) 5.12e20        | <code>scanf( "%Lg", &amp;d );</code> |
| (e) -3000000033    | <code>scanf( "%li", &amp;m );</code> |
| (f) 5,000,000,033  | <code>scanf( "%li", &amp;m );</code> |
| (g) -3000000033    | <code>scanf( "%li", &amp;d );</code> |
| (h) 333222111000.9 | <code>scanf( "%lg", &amp;d );</code> |

4. Which operation (integer division or real division) will be used to evaluate each of the following divisions? Assume that `h`, `k`, and `m` are type `int` while `x` is type `double`.

- (a) `k = h / 3;`
- (b) `k = 3.14 / m;`
- (c) `x = h / m;`
- (d) `k = h / x;`
- (e) `h = x + k / m;`
- (f) `h = k + x / m;`

5. Define the following and give an example of code that might cause it:

- (a) Integer overflow error
- (b) Floating-point underflow error
- (c) NaN error
- (d) Precision error

6. Show the output produced by the following program. Be careful about spacing.

```
#include <stdio.h>
int main(void)
{
    int i = 0;
    float x = 1.2959;

    while (i < 4) {
        printf( "%6.2e %6.2f %6.2g \n", x, x, x );
        x *= 10;
        i++;
    }
    return 0;
}
```

7. For each computation that follows, say whether overflow will occur if integers are 2 bytes long? If they are 4 bytes long?

```
int J, K=100, M=2000;
```

- (a) `J = K * K * K;`
- (b) `J = (float)K * K * K;`
- (c) `J = 30 * M / K * M;`
- (d) `J = M * M * M;`

8. Say whether each computation that follows will give a meaningful answer or is likely to cause overflow, underflow, or serious precision error.

```
float c = 80000;
float d = 1.0e-5;
float f;
```

- (a)  $f = \text{pow}(c, 5) / d;$
- (b)  $f = \text{ceil}(d) * 2e-90;$
- (c)  $f = d + \text{sqrt}(100 * c);$
- (d)  $f = \text{sqrt}(10 * d);$

9. Show how the following normalized numbers would look when printed in `%.3f` format:

- (a) `3.245E+02`
- (b) `1.267E-03`
- (c) `3.14E+04`
- (d) `1.02E-03`

10. Several problems are associated with doing calculations with real values. Which of these do you believe occurs most often? Which of these, even if it does not occur often, causes the most trouble and why?
11. Something is wrong with the following tests for equality. For each item, explain why the answer will be different from the intended answer.

```
short int s=1, t=32767;
long int k=65536;
float w=3.3;
double x=3.3, y=33.0;
```

- (a) `x == w`
- (b) `x*10.0 == y`
- (c) `s == (short)k`
- (d) `32769 == (int)t + 2`

### 7.7.3 Using the Computer

1. Summation.

A simple mathematical function can be defined by the equation

$$f(N) = \sum_{x=1}^N x \sin(x)$$

as  $x$  increases from 1 to  $N$  degrees in 1-degree increments. This equation will sum  $N$  terms, each of which multiplies  $x$  times a value of the `sin()` function. Write a function with a parameter  $N$  that will print a table of the  $N$  terms and return the value of  $f(N)$ . Write a main program that will input a value for  $N$ , then call the function  $f(N)$ , and print the result. Check to make sure that the value of  $N$  is positive. If not, give the user another chance to enter a valid value, until it is proper. Remember that the `sin()` function requires the angle to be in radians rather than degrees.

2. Bridge hands.

A bridge deck has 52 cards and a hand consists of 13 cards. Calculate the following facts about possible bridge hands. Use the information in Section 7.5.4 and Figure 7.27 as guidance.

- (a) How many possible different bridge hands are there? (Call this number  $H$ .)
- (b) How many hands include all four of the Aces in the deck? The formula is:

$$A = \frac{48!}{9! \times 39!}$$

- (c) What is the probability of receiving a hand that has all four aces? The formula is:  $A/H$ .

- (d) Do any of the above computations require special care when done with type `float`? Explain why or why not.
3. See your money grow.  
 Assume you are loaning money to a friend, who will pay it back as a lump sum at the end of the loan period, with interest compounded monthly. Write a program that will allow you to enter an amount of money (in dollars), a number of months, and an annual interest rate. From these data, first calculate a monthly interest rate (1/12 of the annual rate). Then print a table with one line per month, showing the month number, the amount of interest your money will earn that month, and the total amount of your investment so far after the interest is added. Print one line per month, from the time the loan is made until the time it is repaid. Print column headings and print all values in neat columns under them. Break your output into readable blocks by printing a blank line after every twelfth month. Be sure to test your program with a loan period greater than 12 months.
4. Loan payments.  
 Compute a table that shows a monthly payback schedule for a loan. The principle amount of the loan, the annual interest rate, and the monthly payment amount are to be read as inputs. Calculate the monthly interest rate as 1/12 of the annual rate. Each month, first calculate the current interest = the monthly rate  $\times$  the loan balance. Then add the interest amount to the balance, subtract the payment, and print this new balance. Continue printing lines for each month until the normal payment would exceed the loan balance. On that month, the payment amount should be the remaining balance and the new balance becomes 0. Print a neat loan repayment table following this format:

```
Payment schedule for $1000 loan
at 0.125 annual interest rate
and monthly payment of $100.00
```

Month	Interest	Payment	Balance
1	10.42	100.00	910.42
2	9.48	100.00	819.90
...	...	...	...
10	1.66	100.00	60.99
11	0.64	61.63	0.00

5. Bubbles.

The internal pressure inside a soap bubble depends on the surface tension and the radius of the bubble. The surface tension is the force per unit length of the inner and outer surface. The equation for the pressure inside the bubble relative to the air pressure outside is

$$P = \frac{4\sigma}{r} \quad (\text{lb}/\text{ft}^2)$$

where  $\sigma$  is the surface tension (lb/ft) and  $r$  is the bubble radius (ft).

Define a function, `bubble()`, that will compute the pressure  $P$  given a value of  $r$  and assuming the constant  $\sigma$  to be 0.002473 lb/ft. Then write a main program that will input a value for  $r$ , call the `bubble()` function to compute the pressure, convert the units of pressure from  $\text{lb}/\text{ft}^2$  to psi (pounds per square inch,  $\text{lb}/\text{in}^2$ ), and print the answer. Make sure that the input radius is valid; that is, greater than 0. Allow the user to continue entering values until the radius is valid.

6. How functions grow.

Write a program that will ask the user to enter an integer,  $N_{max}$ , then print a table like the following one, with  $N_{max}$  lines. If  $N_{max}$  is less than 1 or more than 20, print an error comment and ask the user to reenter  $N_{max}$ . Store the result of all calculations in variables of type `int`. Use the `pow()` function in the math library to calculate  $2^N$ . The C system will coerce the `double` result of `pow()` to an integer for you. Are all the results correct when you use  $N = 20$ ? If not, why not?

<code>N</code>	<code>sum(1..N)</code>	<code>N squared</code>	<code>2 to the power N</code>
----------------	------------------------	------------------------	-------------------------------

1	1	1	2
2	3	4	4
3	6	9	8
...	...	...	...

## 7. Fibonacci numbers.

A Fibonacci sequence is a series of numbers such that each number is the sum of the two preceding numbers in the sequence. For example, the simplest Fibonacci sequence is: 1, 1, 2, 3, 5, 8, 13, 21, ... In this sequence, the first two terms are, by definition, 1. Write a program to print the terms of this sequence in five columns, as follows:

0. 1	1. 1	2. 2	3. 3	4. 5
5. 8	6. 13	7. 21	...	

Hint: Consider having three variables in your loop, called `current`, `old`, and `older`. After computing the new current value, shift the old values from one variable to the next to prepare for the next iteration. Run your program and determine experimentally how many terms of the Fibonacci series can be computed on your machine before an overflow if you use variables of type `short`, `long`, `float`, and `double` to hold the results.

## 8. Square root.

Over 2000 years ago, Euclid invented a fast, iterative method for approximating the square root of a number. Let  $N$  be a positive number and  $est$  be the current estimate of its square root. (Initially, let  $est = N/2$ .) At each step of the iteration, let  $quotient = N/est$ . If  $quotient$  equals  $est$ , they are the square root of  $N$  and the iteration should end. Otherwise, let the new  $est$  be the average of  $quotient$  and the old  $est$  and repeat the calculation until  $quotient$  equals  $est$  within some epsilon value. Print a table showing the iteration number and the current values of  $est$  and  $quotient$ . Let the user enter the values of  $N$  and epsilon. Then print the value calculated using the standard `sqrt()` function. Run your program several times with epsilon equal to 0.01, 0.001, 0.0001, and so on. Summarize your results in a neat chart with columns for epsilon, the approximation for  $\sqrt{x}$ , and the number of iterations needed to converge with that value of epsilon.

## 9. A table.

Write a program that will ask the user to enter a real number,  $N$ , then print a table showing how certain functions grow as  $N$  doubles. For  $N = 3.14$ , the output should start thus:

	N	1/N	N * log(N)
1	3.14	3.184713e-01	3.592860e+00
2	6.28	1.592357e-01	1.153868e+01
...	...	...	...

Let all your variables be type `float`. Continue computing and printing lines until an underflow occurs in the column for  $1/N$  and an overflow occurs in the last column. Use the `log()` function, which computes the natural log of a number, and use the constant `HUGE_VAL` from the `math` library to test for an overflow. Remember that a value becomes 0.0 when an underflow occurs.



# Chapter 8

## Character Data

This chapter is concerned with the character data type. We show how characters are represented in a program and in the computer; how they can be read, written, and used in a program; and how they are related to integers.

### 8.1 Representation of Characters

A character is represented using a single byte (8 bits). We have shown how numeric values (integers and reals) are represented using the binary number system. Characters also are represented by bits. However, when we think of a text character, a binary number is not the first thing that comes to mind. There is no obvious way in which numbers or bit patterns correspond to the letters, digits, and special characters on a keyboard. So people have invented arbitrary codes to represent these characters. The most common of these is the ASCII (American Standard Code for Information Interchange) **code**, which is listed in a table in Appendix A. Each ASCII character is represented by 7 bits,<sup>1</sup> which are stored on the rightmost side of a byte. You can think of the value of this byte either as a character or an integer.

The ASCII characters are listed in the appendix in numeric order, according to the value of their bit representations. We can find a character in the table and see its code or use a numeric code as an index into the table to determine the associated character. For historical reasons, the indexing number often is listed in two forms, decimal and hexadecimal.<sup>2</sup>

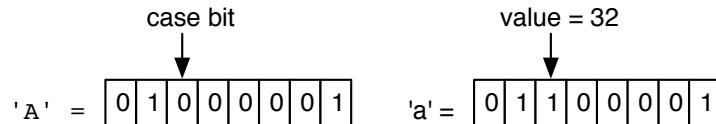
The ASCII codes from 33 to 126 represent printable characters; most of these are letters of the alphabet in upper or lower case. Note that each lower-case letter is 32 greater than the corresponding upper-case letter, for example, 'A' + 32 == 'a'. A single bit in the representation, called the *case bit*, makes this difference. This is illustrated in Figure 8.1

<sup>1</sup>International ASCII uses 8 bits to represent each character and Unicode uses 16 bits.

<sup>2</sup>The hexadecimal number system is discussed in Chapter 15 and in Appendix E.

---

In the ASCII code, upper and lower case letters differ by only one bit, the sixth when counting from the right. This bit has the binary value of 32.



---

Figure 8.1. The case bit in ASCII.

Data type	Define Constant Limits	Range
<code>signed char</code>	<code>SCHAR_MIN..SCHAR_MAX</code>	<code>-128...127</code>
<code>unsigned char</code>	<code>0..UCHAR_MAX</code>	<code>0...255</code>
<code>char</code>	<code>CHAR_MIN..CHAR_MAX</code>	Same as <code>signed</code> or <code>unsigned char</code>

Figure 8.2. Character types.

### 8.1.1 Character Types in C

In reality, characters are just very short integers in C; the single byte of a character holds a number. Anything you can do with an integer, you can do with a character. Anything you can do with a character, you can do with 1 byte of an integer. *There is no difference between a character and an integer* except the number of bytes used and the format specifiers used to read and print the two types.

### 8.1.2 The Different Interpretations

Figure 8.2 lists the character types defined by the C standard. The type `signed char` is not used often, and when used, it is generally thought of as a very short `int`. The more common type, `unsigned char`, is useful primarily when a program must store large quantities of small positive integers in memory and conserve storage to avoid running out of it. In this case, the `unsigned char` actually is being used as a very short `unsigned int`. A program that does image processing is an example of such an application (see Chapter 18). The type `char` is used for most character-handling applications.

A character code such as ASCII uses a fixed number of bits to represent the letters of the alphabet, numerals, punctuation marks, special symbols, and control codes. ASCII uses 7 bits and therefore can represent 128 codes. The C standard permits type `char` to be defined either as signed or unsigned values; the compiler manufacturer makes that decision. Normally, it is of no concern to the programmer, since the index values for the ASCII table (0–127) are present in both forms, so there is not such a **portability** problem as there is with `int`. However, an international version of the ASCII code uses all 256 index values. The extra codes are used to represent additional letters and special symbols used in various European languages. In systems that use International ASCII, `char` is implemented as an unsigned type.

### 8.1.3 Character Literals

Most often **character literals** are written in C using single quotes, like this: '`A`'. However, nothing in C is simple. The character literal inside the quotes can be written two ways, as shown in Figure 8.3:

- If it is an ordinary printable character, we write it directly in quoted form. Thus, the first letter of the alphabet is written as '`a`' (lower case) or '`A`' (upper case).
- Some characters are written with an **escape code** or escape sequence. This consists of the `\` (escape) character followed by a code for the character itself. The predefined symbolic escape codes are listed in Figure 8.4, a few of which we already used in output formats.

Escape code characters are included in C for two different reasons: to resolve ambiguity and to provide visible symbols for invisible characters. Three of the escape codes, `\'`, `\\"`, and `\\"`, are used to resolve lexical and syntactic ambiguity. The backslash (also called *escape*), single quote, and double quote characters have special meaning in C, but we also need to be able to write and process them as ordinary characters. The escape character tells C that the following keystroke is to be treated as an ordinary character, not as an element of C syntax.

The other escape characters are invisible; their purpose is to cause a side effect. For example, the *attention* code, `\a`, is used to alert the user that something exceptional has happened that needs attention. (Note that '`\a`' and '`a`' are very different; the first means "attention" and should cause most computers to beep; the second is an ordinary letter.) A very important escape code is the *null character*, `\0`, which is used to mark the end of every character string and will be discussed further in Chapter 12.

One set of escape code characters that we use frequently are called **whitespace characters**; they affect the appearance of the text but leave no visible mark themselves. The list includes newline, `\n`; return, `\r`;

---

Character constants can be written symbolically or numerically. The symbolic form is preferable because it is portable; that is, it does not depend on the particular character code of the local computer.

Meaning (Portable)	Symbol	Decimal Index (ASCII Only)
The letter A	'A'	65
Blank space	' '	32
Newline	'\n'	10
Formfeed	'\f'	12
Null character	'\0'	0

---

Figure 8.3. Writing character constants.

horizontal tab, \t; vertical tab, \v; formfeed, \f; and the ordinary space character. The two tab characters insert horizontal spaces or vertical blank lines into the output (the precise number of horizontal or vertical spaces depends on the system). Whitespace characters often are treated as a group in C and handled specially in a variety of ways.<sup>3</sup> Note that whitespace characters are all invisible, but many invisible characters, including null and attention, are not classified as “whitespace”.

## 8.2 Input and Output with Characters

Character input and output can be performed using the standard `scanf()` and `printf()` functions. In addition, other special functions exist just for characters. Some of these functions have subtle difficulties associated with them, which we discuss.

### 8.2.1 Character Input

The standard library functions for reading characters are

1. `getchar()`. This function has no parameters. It reads a single character of input and returns it as an `int`. The character is stored in the rightmost part of that `int`, and bytes to the left of the character are filled with **padding** bits. When the padded value is stored in a character variable, the padding is discarded. This process of adding and stripping off padding is automatic, and transparent, and can be ignored by beginning programmers. Normally, the value returned by `getchar()` is used in an assignment statement.

Example: `ch = getchar();`

2. `scanf()` with a "%c" conversion specifier. In this format, there is *no space* between the opening quotation mark and the %c specifier. This will read the next input character, whether or not it is whitespace, and store it in the address provided. This version is equivalent to using the `getchar()` function.

Example: `scanf( "%c", &ch );`

---

<sup>3</sup>These ways will be explained as they become relevant to the text.

Code	Meaning	Code	Meaning	Code	Meaning
\0	Null	\a	Attention	\"	Double quote
\n	Newline	\b	Backspace	\'	Single quote
\r	Return	\f	Formfeed	\\"	Backslash (escape)
\t	Horizontal tab	\v	Vertical tab		

---

Figure 8.4. Useful predefined escape sequences.

3. **`scanf()` with a "%c" conversion specifier.** In this format, there is a space in the format string before the `%c` specifier. The space causes `scanf()` to skip leading whitespace (if any exists) before reading a single nonwhitespace character and storing it in the address provided. This is similar to the manner in which other data types are scanned.

Example: `scanf( " %c", &ch );`

**Keyboard input is buffered.** Whether you are entering numeric or character data into a program, your input is not sent immediately to the program. Until you hit the Enter key, it is displayed on the screen but remains in a holding tank called the **keyboard buffer** so that you can inspect and change it, if necessary. It is not the case that as soon as you type a character the program will read it and begin processing. Some languages provide this feature, but C is not one of them.<sup>4</sup> After you hit Enter, your input moves to another area called the **input buffer** and becomes available to the program. The program will read as much or as little as called for by the `scanf()` or `getchar()` statement. Unread data remain in the input buffer and will be read by future calls on `scanf()` or `getchar()`.

**Whitespace characters complicate input.** When reading integers or floating-point numbers, `scanf()` skips over leading whitespace characters and starts reading with the first data character. However, with the `"%c"` specifier, leading whitespace is not ignored. If the first unread character is whitespace, that is what the system reads and returns. The function `getchar()` does the same thing. It reads a single character, which might be whitespace. Reading data in this manner leads to surprising behavior if the input contains unexpected whitespace characters such as `\n` and `\t`. Since these are not visible, it is easy to forget that they may be present.

In any text-processing program, it is frequently necessary to skip over indefinite amounts of whitespace. As mentioned previously, the behavior of `scanf()` can be changed by adding a single blank to the format: `" %c"`. The space inside the quotes and before the `%c` tells `scanf()` to skip over leading whitespace, if any exists. However, there is no way to force `getchar()` to skip over these invisible characters. For this reason, we usually use `scanf()` rather than `getchar()` to input single characters interactively.

### 8.2.2 Character Output

Character output is relatively straightforward. The `stdio` library provides two ways to display or print a single character:

1. **`putchar()`.** When only one character of output is needed, `putchar()` is the easiest way to do the job; we simply pass it the character we want to display. For example, to move the screen cursor to the beginning of the next line, we might say `putchar( '\n' );`. Note that the `putchar()` function does not automatically move the cursor to the next line as `puts()` does.
2. **`printf()` with a %c conversion specifier.** When printing a character mixed in with other kinds of data, we use `printf()` with the `%c` format specifier. Example:

```
printf( "Child is %c, %i years old.", gender, age );
```

Of course, a specific (nonvariable) character can be included in the format string itself. As with the other data types, it is possible to specify a field width between the `%` and the `c`, and the printed character will be right or left justified in the field area, depending on the sign of the width specifier.

Since characters *are* integers, it is legal to read or print them as integers. When you read an integer that is the ASCII code of a letter and print that number using a `"%c"` format or `putchar()`, you see the letter. Conversely, when you read a letter and print it using a `"%i"` format, you see a number.<sup>5</sup> This technique is demonstrated in Figure 8.5.

#### Notes on Figure 8.5. Printing the ASCII codes.

---

<sup>4</sup>Some old PC-based single-user C systems support the unbuffered character input functions `getch()` and `getche()`, in a library named `conio`. When using `getch()`, any character that the user typed would go directly to the program, rather than being held in the keyboard buffer until a newline was typed. This seems like a convenient function and it was popular with students. It is not supported by the standard because modern systems are multi-processing systems and cannot make a direct connection between any input device and any one process among the set that is running concurrently. We recommend against using any nonstandard feature because it is not portable.

<sup>5</sup>It also is possible to print the hexadecimal form of the character's index by using a `%x` conversion specifier. See Chapter 15.

---

```
#include <stdio.h>
int main( void )
{
    char ch;

    puts( "\n Demo: Printing the ASCII codes." );
    printf( "\n Please type a character then hit ENTER: " );
    ch = getchar();

    printf( " The ASCII code of %c is %i \n\n", ch, ch );
}
```

---

**Figure 8.5.** Printing the ASCII codes.

**First box: declaration.** We declare a `char` variable. In the remaining code, we use it to perform both character and integer output.

**Second box: character input.** We read a character (one keystroke); its ASCII code is stored as a binary integer in the `char` variable. The character is not read until the Enter key is pressed.

**Third box: output.** We print the input character twice: first as a character, using `%c`; then as an integer, using `%i`. Sample program output looks like this:

Demo: Printing the ASCII codes.

```
Please type a character then hit ENTER: A
The ASCII code of A is 65
```

### 8.2.3 Using the I/O Functions

The program in Figure 8.6 illustrates the use of the four character input and output functions and demonstrates how a simple program can produce very confusing output if the problem of whitespace in the input is not addressed.

#### Notes on Figure 8.6. Character input and output.

**First box: character output.**

- The function `putchar()` prints a single character. Its argument can be a character variable, a literal character, or an integer. If the argument is an `int`, the rightmost byte of its value is interpreted as an index for the ASCII code table, and the character in that position is printed. The command `putchar( 42 )` prints an asterisk because 42 is the ASCII code for `*`.
- We also can print a single character using `printf()` with `%c`; in this case, we print a dollar sign. If we try to print an integer with a `%c` conversion specifier, we see the character that corresponds to that integer's index in the code table. For example, the output from `printf( "%c", 42 )` would be an `*`.

**Second box: reading the first response.**

- The line `scanf( "%c", ... )` reads a single character of input.
- When a program begins executing, the input buffer is empty. Normally, the user will not enter any input until prompted, so the user's response to the first prompt will be the only thing in the input buffer. The first character of that response will be read, while the newline character generated by the Enter key will remain in the buffer. We presume that the user will type `y`, causing control to enter the loop.

**Third box: the loop.**

- In this loop, we “give” \$5.00 to the user and prompt for another response:

Here is \$5.00

Do you need more (y/n)?

---

We demonstrate various ways to read and write single characters and how a whitespace character in the input can cause unexpected results.

```
#include <stdio.h>
int main( void )
{
    char input;
    char money = '$';
    char star = '*';

    putchar( '\n' );  putchar( star );  putchar( 42 );  putchar( '\n' );
    printf( " Do you need %c (y/n)? ", money );

    scanf( "%c", &input );

    while (input == 'y') {
        printf( " Here is %c5.00\n", money );
        printf( "\n Do you need more (y/n)? " );
        input = getchar(); /* This code is wrong! Use scanf( " %c", &input ) */
    }

    printf( " OK --- Bye now. %c \n", '\a' );
}
```

---

**Figure 8.6. Character input and output.**

- This time we use `getchar()` to read the next input character. If it is `y`, we will stay in the loop; for any other input (including whitespace), we leave the loop.
- This logic seems simple enough, but *it does not work*. As shown in the following output, the user sees the second prompt but the program quits and says goodbye without giving that user a chance to enter anything. The reason for this problem is explained in the following section.
- The complete output for this run is

```
**
Do you need $ (y/n)? y
Here is $5.00

Do you need more (y/n)? OK --- Bye now.
```

**Fourth box: the closing message.** The `%c` “prints” the escape character `\a`. On some systems, if the computer has a sound generator and the volume is turned up, you should hear a ding when you print the attention character, but you see nothing. On other systems, the output may be visible (for instance, a small box) but not audible.

**Problems with `getchar()`.** A common programming error was illustrated in Figure 8.6. After `scanf()`, the program initially performs as expected; if the input is `y`, it enters the loop and “gives” the user `$5.00`. Then the loop prompts the user to enter another `(y/n)` response. However, when the second prompt is displayed, the system does not even wait for the user to respond; it simply quits. Why?

When entering the answer to the first question (above the loop), the user types `y` and hits the Enter key. This puts the character `y` and a newline character into the input buffer. The newline character is necessary because, in most operating systems, the system does not send the keyboard input to the program until the user types a newline. The `scanf()` above the loop reads the `y` but leaves the `\n` in the buffer. At the end of the first time around the loop, that `\n` still is sitting in the input buffer, unread. The loop prompts the user for a

choice, but the program does not wait for a key to be hit because input already is waiting. The `getchar()` then reads the `\n`, emptying the buffer. Since `\n` is not equal to `y`, the loop ends and the program says goodbye.

Now, if the user had typed `yyy` followed by a newline instead of a single `y` at the first prompt, the characters waiting in the input buffer would have been `yy\n`. The input to the second prompt then would have been `y`, and the program would have given the user another \$5.00 bill. Altogether, the user would get three bills before the program could read the newline and leave the loop, all with no further typing by the user. The resulting output would be

```
**
Do you need $ (y/n)? yyy
Here is $5.00

Do you need more (y/n)? Here is $5.00
Do you need more (y/n)? Here is $5.00

Do you need more (y/n)? OK --- Bye now.
```

Using `scanf( "%c", &input )` in place of `input = getchar()` does not solve the problem, because `scanf()` with `"%c"` works the same way as `getchar()`. However, we can solve this whitespace problem by using a single space in the format for `scanf()`. Replace the call on `getchar()` in Figure 8.6 by this call on `scanf()`:

```
scanf( " %c", &input );
Note space in format
```

With this change, everything will work as intended: The program will query the user, wait for a response every time, and do the appropriate thing. A typical output would look like this:

```
**
Do you need $ (y/n)? y
Here is $5.00

Do you need more (y/n)? y
Here is $5.00

Do you need more (y/n)? n
OK --- Bye now.
```

If the user initially were to type `yyy`, the output would begin as shown earlier, but after three times through the loop, the user would have a chance to enter responses again.

#### 8.2.4 Other ways to skip whitespace.

Skipping whitespace by inserting a space into a format specifier is a little-known technique, even though it is easy to do and easy to understand. A common mistake among programmers is to use the C function `fflush()` to do this task. The C standard states that *the function `fflush()` is defined only for output streams*, not for input. Sometimes it seems to work for input, but it does so for indirect and subtle reasons, and only in some implementations. A correct alternative technique for skipping whitespace is to use a simple loop, as shown in Figure 8.7.

##### Notes on Figure 8.7. A function for skipping whitespace during keyboard input.

**Line 1: The variable declaration.** We declare an `int` variable to store the character being read. Although this seems wrong, remember that the function `getchar()` returns an `int`, not a `char`.

---

```
void skip_ws( void ) {
    int ch;                                // Most recently read character.
    while (isspace( ch=getchar() ) ); // Tight loop; exit when non-space is read.
    ungetc( ch, stdin );                  // Put non-space character back into stream.
}
```

---

**Figure 8.7.** A function for skipping whitespace.

**Line 2: Reading and testing the data.**

- The function `isspace()` is explained in Section 8.3.5. Briefly, it returns `true` if its argument is a whitespace character, `false` otherwise.
- This line is a “tight loop”; it has no body at all and does nothing except read and test, read and test, until the input is non-whitespace.
- This loop will work correctly for any combination and any number (zero or more) of whitespace characters. It will just keep reading until the user types something else. (Remember that many invisible characters are not classified as “whitespace”.)
- We store the input character in a variable because we will use the final character read after the end of the loop.

**Line 3: Get and give back.** We leave the `getchar()` loop when a non-whitespace character is found. However, that is too late! That character is real input and cannot be processed in this function. The easiest remedy is to put that character back into the input stream so that it can later be read by the proper part of the program. Happily, C supplies a function for this task: `ungetc()`. The arguments in this call are the character that must be put back into the input stream, and the name of the input stream. This function will be explained more fully in the chapter about streams and files, Chapter 14.

**A useful tool.** This is a function that will often be useful in programs that analyze character input. Copy the definition into your personal file of C-tools.

## 8.3 Operations on Characters

### 8.3.1 Characters Are Very Short Integers

The basic operations defined for characters are the same operations that are defined for integers because, technically, characters *are* integers in the range 0...255 or -128...127. Some kinds of data (such as digital images) are composed of a very large number of very small integers. In such cases, it is useful to minimize the amount of storage occupied by the data, so the data are stored as type `char` or `unsigned char` rather than `short int` or `int`. In such applications, it is important that all the integer operations can be applied to variables of type `char`.

The common use of type `char`, however, is to represent characters. Even then, many integer operators are useful; These are summarized in Figure 8.8. A little caution is warranted here; some integer operations are legal but not useful with characters. For example, it makes no sense to multiply or divide one character by another. Unfortunately, useless or not, the compiler will not identify such expressions as errors. In addition to the basic integer operators, there is also a set of functions in the `ctype` library that can manipulate character values. We now examine some of the library functions and take a closer look at the operators in Figure 8.8.

### 8.3.2 Assignment

We have seen that the values of both character and integer variables can be assigned to a `char` variable. As long as the integer value is not too big, everything will be fine (large values lead to overflow). Automatic coercion will shorten the integer and a single byte will be assigned. Literal characters also can be assigned to `char` variables. The columns of Figure 8.3 show two different values that will assign the same character code to a variable.

Operation	Meaning and Use
<code>char c1, c2;</code>	Declare two character variables
<code>c1 = c2;</code>	Copy the value of <code>c2</code> into <code>c1</code> .
<code>c1 == c2</code>	Do <code>c1</code> and <code>c2</code> contain the same letter?
<code>c1 != c2</code>	Do <code>c1</code> and <code>c2</code> contain different letters?
<code>c1 &lt; c2</code>	Does <code>c1</code> come before <code>c2</code> in alphabetical order?
<code>c1 + 1</code>	The letter that follows the value of <code>c1</code> in the alphabet.
<code>c1 - 1</code>	The letter that precedes the value of <code>c1</code> in the alphabet.
<code>++c2;</code>	Change <code>c2</code> from its current value to the next letter in the alphabet. All four increment and decrement operators are defined for characters.
<code>c2 - c1</code>	Assuming that <code>c2 &gt; c1</code> , this is the number of letters in the alphabet between <code>c1</code> and <code>c2</code> . If <code>c2</code> is the ASCII code for a base-10 digit, then <code>c2 - '0'</code> is the numeric value of that digit.

Commonly used character operations are listed here. The phrase *alphabetical order* used here means “the order defined by the ASCII code or whatever code is in use on the local hardware.” This normally is an extension of ordinary alphabetical order to include all of the characters in the local character set.

Figure 8.8. Character operations.

### 8.3.3 Comparing Characters

The operators `==` and `!=` are used to test whether two characters are equal. These operators are straightforward and portable; that is, they work identically on all systems. The other four comparison operators, `<`, `>`, `<=`, and `>=`, also are useful for characters, but their results can vary because they depend on the local computer system.

ASCII and International ASCII are the two codes used most commonly in personal computers today, but some systems use different underlying character codes. The particular code in use on the local system determines the “alphabetical order” on that system. (The technical terms for alphabetical order are **collating sequence** and **lexical order**.) Numerals and letters of the English alphabet are arranged in the usual order in most codes, but the special symbols may be arranged in arbitrary and incompatible ways. Therefore, two dissimilar machines might produce different results for some character comparisons.

### 8.3.4 Character Arithmetic

The operators `+`, `-`, `++`, and `--` are used in **character arithmetic** to compute the next (or prior) letters of the alphabet. The operator `-` can be used to determine how far apart two letters are in the alphabet. All these operations are useful for text processing programs and all depend on the collating sequence of the machine.

### 8.3.5 Other Character Functions

One of the standard C libraries is the character processing library, whose header file is `ctype.h`. This library contains a group of functions essential to a system programmer, including ones to test whether a character is in a particular set, such as the alphabet, as well as certain transformation routines. Several of these functions are frequently useful, even in simple programs:

1. `isalpha()`. This function takes one argument, a character. If the character is an alphabetic character (A ... Z or a ... z), the value `true` (1) is returned; otherwise, `false` (0) is returned.
2. `islower()`. This function takes one argument, a character. If the character is a lower-case alphabetic character (a ... z), the value `true` (1) is returned; otherwise, `false` (0) is returned.
3. `isupper()`. This function takes one argument, a character. If the character is an upper-case alphabetic character (A ... Z), the value `true` (1) is returned; otherwise, `false` (0) is returned.

4. **isdigit()**. This function takes one argument, a character. If it is a digit (0...9), **true** (1) is returned; otherwise, **false** (0) is returned.
5. **isspace()**. This function takes one argument, a character. If the character is a whitespace character (space, newline, return, formfeed, horizontal tab, or vertical tab), **true** is returned; otherwise, **false** is returned. We use **isspace()** to help analyze input data so that the results will be the same whether the user types spaces, newlines, or tab characters.
6. **tolower()**. This function takes one argument, a character. If the character is an upper-case alphabetic character (between A and Z), the return value is the corresponding lower-case character (between a and z). If the argument is anything else, it is returned unchanged. This function sets the case bit to 1.
7. **toupper()**. This function is the opposite of **tolower()**. If the argument is a lower-case character, the return value is the corresponding upper-case character. If the argument is anything else, it is returned unchanged. This function sets the case bit to 0.

When an input buffer might contain an unpredictable sequence of input items, a program can use the functions **isalpha()**, **islower()**, **isupper()**, **isdigit()**, and **isspace()** to analyze each input character and handle it appropriately. This makes it easier to build a well-designed user interface.

The functions **toupper()** and **tolower()** often are used in conjunction with testing character input, so that it does not matter whether the user types an upper-case or lower-case character. The programmer chooses one case to use for processing (often, it does not matter which) and converts all input characters to that case. By doing so, the logic of the program can be simplified. For example, suppose that a program needs to read a character into a variable named **ch** and use it to select one action from a set of actions. We could write the code like this:

```
scanf( " %c", &ch );
if (ch == 'x' || ch == 'X') { /* X actions here */ }
else if (ch == 'y' || ch == 'Y') { /* Y actions here */ }
else if (ch == 'z' || ch == 'Z') { /* Z actions here */ }
else { /* default actions */ }
```

By using a case-shift function, we can eliminate one comparison in each test:

```
scanf( " %c", &ch );
ch = tolower( ch );
if (ch == 'x') { /* X actions go here */ }
else if (ch == 'y') { /* Y actions go here */ }
else if (ch == 'z') { /* Z actions go here */ }
else { /* default actions go here */ }
```

## 8.4 Character Application: An Improved Processing Loop

This example combines the use of **scanf( " %c", ... )** with **toupper()**, **tolower()**, and a **switch** with character case-labels.

In Chapter 6, Figure 6.12, we demonstrate how a process can be repeated using a query loop until the user chooses to quit. Using the codes 1 to continue and 0 to quit is adequate, but it is not good human engineering and not customary. Now that we have shown how to read a single input character and how to force that character into a particular case, it is possible to improve the human interface by giving the more usual prompt: **Do you want to continue (y/n)?** and accepting either an upper-case or lower-case answer. The next program implements this improved interface.

The **work()** function, in Figure 8.10, is a simple application that computes the area of a regular polygon or circle. It prompts the user to select the kind of polygon from a menu and uses **tolower()** with a **switch** and character case labels to process that choice.

### Notes on Figure 8.9. Improving the workmaster.

**First box: the #include statements.** In addition to **<stdio.h>**, we must include **<ctype.h>** because we are using the character-function library.

We improve the user interface of the main program from Figure 6.12 by permitting a y/n or Y/N response to the question, “Do you want to continue?” This program calls the `work()` function in Figure 8.10.

```
#include <stdio.h>
#include <ctype.h> /* For toupper() and tolower(). */

void work( void );
double circle_area( double diam ) { return 3.1416 * diam * diam / 4; }
double square_area( double side ) { return side * side; }
double triangle_area( double side ) { return side * side / 4.0 * sqrt( 3 ); }

int main( void )
{
    char more; /* repeat-or-stop switch */

    puts( "\n Calculate the area of a regular figure." );
    do { work();
          puts( "\n Do you want to continue (Y/N)? " );
          scanf( " %c", &more );
          more = toupper( more );
        } while (more != 'N');
    return 0;
}
```

Figure 8.9. Improving the `workmaster`.

**Second box: three functions.** These functions perform the area computations for three different shaped figures. They are called from the `work()` function in Figure 8.10.

**Third box: the char variable.** We use a character variable rather than an integer to store the user’s quit-or-continue response.

**Fourth box: the repetition loop.** We change the `do...while()` termination condition to test for the letter ‘N’ instead of the number 0. This is more natural for the user. However, to make this test work reliably, we need to change two things in the inner box. Either response, ‘N’ or or ‘n’ will cause the loop to end; any other response will permit it to continue.

**Inner box: reading input.** To read the input, we use a `scanf()` format that will skip whitespace in the input buffer. This is important in a loop that controls a `work()` function, because the input operations performed by that function usually leave whitespace (at least one newline character) in the input buffer.

**Innermost box: case conversion.** Even when instructions call for a Y or N response, many users will often type y or n instead. Good human engineering dictates that the program should accept upper-case and lower-case letters interchangeably. We achieve this by using `toupper()` to force the response into upper case. This permits us to make a simple check for an upper-case response instead of the more complex test for either a lower-case or upper-case letter. To achieve the same result without `toupper()`, we would have to write the loop test as

```
while (more != 'n' && more != 'N').
```

**Notes on Figure 8.10. Using characters in a switch.**

**First box: the menu.** We display a simple menu and prompt the user for a choice. When we read the input character, we use a space in the format to skip over the carriage return character left in the input stream by `main()`.

**Second box: using `tolower()` in a switch.** We use `tolower()` here so that the user can enter either upper-case or lower-case choices. This line could be written thus: `switch (ch)`. However, if it were written without the call on `tolower()`, the following case labels would need to be more complex.

This function is called from `main()` in Figure 8.9. Code from both Figures should be in the same file.

```
void work( void )
{
    char ch;      /* Length of one side or of the diameter */
    double x;      /* Length of one side or of the diameter */
    double area;   /* Area of the figure */

    printf (" Enter the code for the shape you wish to calculate: \n" );
    printf (" C Circle\n S Square\n T Equilateral triangle\n > " );
    scanf( " %c", &ch );

    switch (tolower( ch ))
    {
        case 'c':
            printf( " Enter the diameter of the circle: " );
            scanf( "%lg", &x );
            area = circle_area( x );
            printf( " The area of this circle = %.2f\n", area );
            break;

        case 's':
            printf( " Enter the length of one side of the square: " );
            scanf( "%lg", &x );
            area = square_area( x );
            printf( " The area of this square = %.2f\n", area );
            break;

        case 't':
            printf( " Enter the length of one side of the triangle: " );
            scanf( "%lg", &x );
            area = triangle_area( x );
            printf( " The area of this triangle = %.2f\n", area );
            break;

        default: printf( "%c is not a meaningful choice. Try again.", ch );
    }
    return 0;
}
```

Figure 8.10. Using characters in a switch.

**Third box: the case label.** If the call on `tolower()` were omitted in the second box, we would need to write two case labels for each case, like this: `case 'c': case 'C':`

**Fourth box: the case actions.** We perform all of the actions needed for a circle: input, calculation using the appropriate function, and output. Of course, a `break` statement must end the sequence. It is good style to use functions to do much or most of the work for each case, so that the entire `switch` statement fits on one computer screen.

**Fourth box: the default.** This case traps illegal menu choices. You can see the result in the last block of output, below.

**Output.** A sample of the output follows. Note that whitespace and case differences are ignored, and that the invalid response to the second query causes the process to continue, not quit.

```

Calculate the area of a regular figure.
Enter the code for the shape you wish to calculate:
C Circle
S Square
T Equilateral triangle
> c
    Enter the diameter of the circle: 10
    The area of this circle = 78.54

Do you want to continue (Y/N)? : y
Enter the code for the shape you wish to calculate:
C Circle
S Square
T Equilateral triangle
> t
    Enter the length of one side of the triangle: 3.5
    The area of this triangle = 5.30

Do you want to continue (Y/N)? : t
Enter the code for the shape you wish to calculate:
C Circle
S Square
T Equilateral triangle
> s
    Enter the length of one side of the square: 10
    The area of this square = 100.00

Do you want to continue (Y/N)? : y
Enter the code for the shape you wish to calculate:
C Circle
S Square
T Equilateral triangle
> w
    w is not a meaningful choice. Try again.

Do you want to continue (Y/N)? : n

```

## 8.5 What You Should Remember

### 8.5.1 Major Concepts

- ASCII is a character code. It uses 7 bits to represent the set of 128 characters that are part of the code. International ASCII is an 8-bit code that represents 256 characters. These codes are used with the `char` data type.
- Character literals are written between single quotemarks.
- Escape codes are used to write literals for invisible characters.
- There are several whitespace characters, including space, horizontal and vertical tabs, and newline.

### 8.5.2 Programming Style

**Escape codes.** Most ASCII characters are printable characters; that is, they leave a visible mark when displayed on a video screen or a printer. These characters correspond to keys on a typical computer keyboard. Some keys, such as the space bar, the Tab key, and the Enter key, do not represent printable characters but are used for their effect on the printed text. These, called *whitespace* characters, are represented in a C program by symbolic escape codes. There also are nonprintable ASCII characters that have no symbolic escape codes; they are used infrequently but may be referenced, if necessary, by using the underlying value. To be sure that your program is portable, use only the literal form of a character or a symbolic code.

**Avoiding errors.** Use character processing for a better human interface, like that in the revised `work()` function. Use functions like `toupper()` and `tolower()` to handle both upper-case and lower-case responses.

### 8.5.3 Sticky Points and Common Errors

**char vs. int.** Technically, characters are very short integers in C. Conceptually, though, they are a separate type with separate operations and different methods for input and output. Be sure not to do meaningless operations like multiplying two characters and avoid potentially nonportable operations like `<`. Every C implementation uses the character code built into the underlying hardware. For most modern machines, that code is either International ASCII or ASCII, which is given in Appendix A.

**Character input.** Whitespace can be a confusing factor when doing character input. The `scanf()` input conversion process for numeric types automatically skips leading whitespace and starts storing data only when a nonwhitespace character is read. However, `getchar()` returns the first character, no matter what it is; and `scanf()` with a "%c" does the same thing. To skip leading whitespace, you must use `scanf()` with a " %c" specifier (a space inside the format and before the percent sign). If this space is omitted, the program is likely to read whitespace and try to interpret it as data, which usually leads to trouble. Therefore, a programmer must have a clear idea of what he or she wishes to do (read whitespace or skip it) and choose the appropriate input mechanism for the task.

### 8.5.4 New and Revisited Vocabulary

These are the most important terms and concepts presented in this chapter:

character literal	lexical order	padding
escape code	collating sequence	character arithmetic
portability	input buffer	improved <code>work()</code> function
ASCII code	whitespace	<code>switch</code> with <code>char</code> cases

The following C keywords, functions, and symbols are discussed in this chapter:

\n (newline)	<code>getchar()</code>	<code>isalpha()</code>
\r (return)	<code>putchar()</code>	<code>isupper()</code>
\b (backspace)	<code>printf()</code>	<code>islower()</code>
\t (horizontal tab)	<code>scanf()</code>	<code>isspace()</code>
\v (vertical tab)	<code>ungetc()</code>	<code>isdigit()</code>
\f (formfeed)	"%c" conversion	<code>tolower()</code>
\a (attention)	" %c" conversion	<code>toupper()</code>
\0 (null character)	<code>ctype.h</code>	

## 8.6 Exercises

### 8.6.1 Self-Test Exercises

1. Explain the difference between '6' and 6.
2. What is a whitespace character? List three of them. What is an escape code character? List three of them.

3. Show the output produced by the following program. Be careful about spacing.

```
#include <stdio.h>
int main( void )
{
    for (int k = 1; k <= 5; k += 2) {
        printf( "%i:", k );
        putchar( '0'+k );
    }
    putchar( '\n' );
}
```

4. What will be stored in `k`, `c`, or `b` by the following sets of assignments? Use these variables:

```
int k;    char c, d;    int b;
```

- (a) `d = 'b'; c = d+1;`
- (b) `d = 'b'; c = d--;`
- (c) `d = 'E'; c = toupper( d );`
- (d) `d = '7'; k = d - '0';`
- (e) `b = isalpha( '@' );`
- (f) `b = 'A' == 'a';`

5. What will be stored in each of the variables by the input statements on the right, given the line of input on the left. If the combination is an error, say so. Use these variables:

```
int k;    char d;
```

- (a) `a scanf( "%c", &d);`
- (b) `66 scanf( "%c", &d);`
- (c) `70 C scanf( "%i%c", &k, &d);`
- (d) `F scanf( "%i", &k);`
- (e) `go! d = getchar();`
- (f) `\n scanf( "%c", &d);`

6. What is the output from the following program if the user enters Z after the input prompt?

```
#include <stdio.h>
int main( void )
{
    char ch;
    printf( "\n Type a character and hit ENTER: " );
    ch = getchar();
    printf( "%3i %c \n ", ch, ch );
    putchar( ch ); putchar( '\n' );
}
```

## 8.6.2 Using Pencil and Paper

1. What will be stored in `k`, `c`, or `b` by the following sets of assignments? Use these variables:

```
int b, k;    char c, d;
```

- (a) `d = 'A'; k = d;`  
 (b) `d = 'c'; c = toupper( d );`  
 (c) `d = '@'; c = tolower( d );`  
 (d) `k = 66; c = k-1;`  
 (e) `b = isupper( 'A' );`  
 (f) `b = 'A' < 'a';`
2. What will be stored in each of the variables by the input statements on the right, given the line of input on the left. If the combination is an error, say so. Use these variables:

```
int k, m; char c, d;

(a) a      scanf( "%c", &k );
(b) 126    scanf( "%c", &d );
(c) 70 D   scanf( "%i %c", &k, &d );
(d) 70 71   scanf( "%i%i", &k, &m );
(e) U2     scanf( "%c%i", &d, &k );
(f) I 0     c = getchar(); d = getchar();
```

3. Without running the following program, show what the output will be. Use your ASCII table.

```
#include <stdio.h>
int main( void )
{
    int upper = 65;
    int lower = upper + 32;
    int limit = 26;
    int step = 0;
    puts( " Do you read me?" );
    while ( step < limit ) {
        printf( "%2i. %c %c\n", step, upper, lower );
        ++upper;
        ++lower;
        ++step;
    }
    printf( "=====\\n" );
}
```

4. What is the output from the following program if the user enters 80 after the input prompt?

```
#include <stdio.h>
int main( void )
{
    int k;
    printf( "\\n Enter a number 65 ... 126: " );
    scanf ( "%i", &k );
    printf( " %3i %c \\n", k, k );
    putchar( k ); putchar( '\\n' );
}
```

5. Write a code fragment to compare two character variables and print `true` if both are alphabetic and they are the same letter, except for possible case differences. Print `false` otherwise.

### 8.6.3 Using the Computer

#### 1. Character manipulation practice.

Write a program to prompt the user once for a series of characters. Read the characters one at a time using `scanf( " %c", ...)` in a loop. Generate the following output, based on the input, one response per line:

- (a) Echo the input character.
- (b) Call `isalpha()` to find out whether it is alphabetic.
- (c) If so, call `toupper()` to convert it to an upper-case letter and print the result. If not, print an error comment.
- (d) If the character is a period, print a statement to that effect and quit.

#### 2. Volumes.

Write a program to calculate volumes. Start by writing three double→double functions for these three geometric shapes:

- (a) `cylinder()`. The single argument,  $d$ , is the height of a cylinder and also the diameter of its circular base. Calculate and return its volume. The formula is:  $volume = \pi \times d^3 / 4$
- (b) `cube()`. The argument is the length,  $s$ , of one side of a cubical box. Calculate and return its volume. The formula is:  $volume = s^3$
- (c) `sphere()`. The argument is the diameter,  $d$ , of a sphere. Calculate and return its volume. The formula is:  $volume = (4/3) \times \pi \times d^3 / 8$

Write a program that will permit the user to compute the area of several shapes. Use a menu and a switch to process the menu selections. Include a menu item “Q Quit”, and use this instead of a uery loop to end the program.

Read the letter in such a way that whitespace does not matter. Test the input in such a way that upper-case and lower-case differences do not matter. If the letter is `q`, terminate the program. Otherwise, read and validate a real number that represents the size of the figure. If this length is 0 or negative, print an error comment. If it is positive, call the appropriate area function and print the answer it returns. If the letter entered is not `c`, `s`, `t`, or `q`, print an appropriate error message.

#### 3. Palindromes.

This program will test whether a sentence is a palindrome; that is, whether it has the same letters when read forward and backward. First, prompt the user to enter a sentence. Read the characters one at a time using `getchar()` until a period appears. As they are read,

- (a) Echo the input character.
- (b) Call `tolower()` to convert each character to lower case.
- (c) Count the number of characters read (excluding the period).
- (d) Store the converted character in the next available slot in an array.

When a period appears, start from both ends of the array and compare the letters. Compare the first to the last, the second to the second-last, and so forth. If any pair fails to match, leave the loop and announce that the sentence is not a palindrome. If you get to the middle, stop, and announce that the input is a palindrome. Assume that the input will be no more than 80 characters long.

#### 4. Ascending or descending.

Your program should read three numbers and a letter. If the letter is , 'A' or 'a', output the numbers in ascending order. If it is 'D' or 'd', output the numbers in descending order. For any other letter, give an error message and output them in the opposite order that they were read in..

5. Vowels.

Prompt the user to enter a sentence, then hit newline. Read the sentence one character at a time, until you come to the newline character. Count the total number of keystrokes entered, the number of alphabetic characters, and the number of vowels ('a', 'e', 'i', 'o', and 'u'). Output these three counts.

6. Tooth fairy time.

This program will “pronounce” an ordinary sentence with a lisp. Prompt the user to enter a sentence. Read the characters, one at a time, until a period appears. As they are read, convert everything to lower case and test for occurrences of the character 's' and the pair "ss". Replace each 's' or "ss" by the letters 't' and 'h'. Print the converted message using `putchar()`. For example, given the sentence I see his house, the output would be I thee hith houthe.

7. Building numbers.

Write a program that will use a work function to input and process several data sets. Each data set will consist of three input characters if they are all valid base-10 digits, you will convert them to an integer and print it. Otherwise, print an error comment.

- Use `isdigit()` to test for valid inputs.
- Convert the ASCII code for a digit to its corresponding numeric value by subtracting the character '0'. For example, if `ch` contains the character '7', then `ch-'0'` is the integer value 7.
- If the numeric values of your inputs are stored in the variables `a`, `b`, and `c`, then the answer is  $a * 100 + b * 10 + c$

8. A tall story.

Develop a program for a baker who makes wedding cakes. These cakes have multiple layers, and the layers have different shapes. The top layer always is circular, with a diameter of 6 inches. The next layer down is square, each side 7.5 inches long. The third layer would be circular again, with a diameter of 8 inches; and the fourth layer is a 9.5-inch square. The shapes continue to alternate in this pattern and get bigger until the bottom layer is reached. In addition, each layer is 2 inches thick, so the area of its side is  $2'' \times$  the perimeter of the layer. The baker wants to know how much frosting to make for the cake to frost the entire top and side of every layer. Write a program that will read in the number of layers desired for a cake, and then print the total square inches of cake to be covered with frosting. Break up your program as follows:

- (a) Write a function called `surface_area()`. This function has two parameters. One is a character with the value C for circle or S for square. The other is an integer that represents either the diameter for a circle or the length of a side of a square. This function will compute the sum of the top area and the side area of either a circular or a square layer, depending on the character value.
- (b) Write a main program that first will read in the number of layers of the cake. Then it will call the `surface_area()` function for each layer of the cake and total the areas. Finally, print the total.

9. Temperatures.

Write a program and three functions that will convert temperatures from Fahrenheit to Celsius or vice versa. The main program should implement a work loop and use the improved interface of Figure 8.9. The `work()` function should prompt the user to enter a temperature, which consists of a number followed by optional whitespace and a letter (F or f for Fahrenheit, C or c for Celsius). Appropriate inputs might be 125.2F and -72 c. Read the number and the letter, test the letter, and call the appropriate conversion function, described here. Test the converted answer that is returned and print an error comment if the return value is -500. Otherwise, echo the input temperature and print the answer with the correct unit, F or C. (Note that there is no difficulty reading an input in the form 125F; `scanf()` stops reading the digits of the number when it gets to the letter and the letter then can be read by a %c specifier.)

Write two functions: `Fahr_2_Cels()` converts a Fahrenheit argument to Celsius and returns the converted temperature; `Cels_2_Fahr()` converts a Celsius argument to Fahrenheit and returns it. Both functions must test the input to detect temperatures below absolute 0 ( $-273.15^{\circ}\text{C}$  and  $-459.67^{\circ}\text{F}$ ). If an input is out of range, each function should return the special value -500.

10. Try it, I dare you!

Write a program that will display the ASCII code in a table that looks like the one in Appendix A. Be sure not to try to print the unprintable characters directly. Omit the column that lists the hexadecimal codes.



# Chapter 9

# Program Design

In Chapter 5, we introduced the concepts of functions and function calls and defined some basic terminology. In this chapter, we review that terminology, extend the rules for defining and using functions, and formalize many aspects of functions presented in preceding chapters: prototypes, function definitions, function calls, how these elements must correspond, and how the necessary communication actually happens. The concepts of local, global, and external names are presented.

We discuss modular organization and the ways that parts of a modular program must relate to each other. The process of designing a modular program is described and illustrated with a programming example.

## 9.1 Modular Programs

When a program has only 20 to 50 lines, a programmer can keep the entire program structure in mind at once. Many programs, though, have thousands of lines of code. To deal with this complexity, it is necessary to divide the code into relatively independent modules and consider each module in isolation from the others, usually with a main program in one module and groups of closely related functions in others. Each module is composed of functions and declarations that relate to one identifiable phase of the overall project. This is the way professionals have been able to develop the large, sophisticated systems that we use today.

A **module** is a pair of files (a `.c` file and its matching `.h` file) containing programmer-defined types, data object declarations, and function definitions. The order of these parts within the module is quite flexible; the only constraint is that *everything must be declared before it is used*. The modules themselves and the functions within them serve several purposes: they make it easy to use code written by someone else or reuse your own code in a new context. Far more important, though, is that they permit us to break a large program into manageable pieces in such a way that the interface among pieces is fixed and controllable. Functions, their prototypes, and header files make this possible in C; class definitions make it easier in C++.

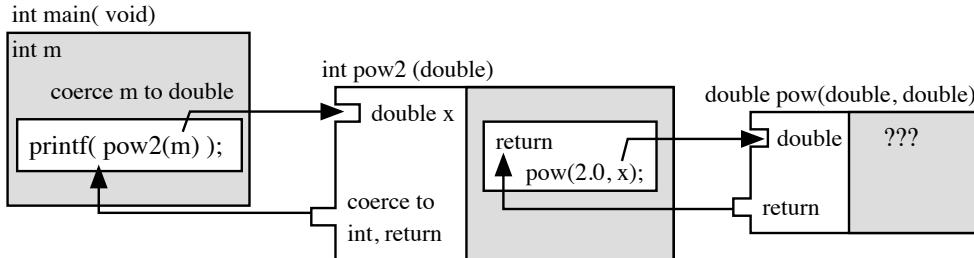
Each **function** is a block of code that can be invoked, or called, from another function and will perform a specific, defined task. All its actions should be related and work at the same level of detail. For example, some functions “specialize” in input or output. Others calculate mathematical formulas or do error handling. No function should be very long or very complex. Complexity is avoided by calling other functions to do subtasks.

One guideline for good style is to keep each function short enough that its parameters, local variable declarations, and code will fit on the video screen at the same time. This limits functions to about 20 lines of code on many computers. Following this guideline, any moderate-sized program will have many functions that are organized into several code modules or classes, with each module or class in a separate source file. The question then arises of where various objects should be declared: in which module and where within the module. This level of design is touched on in Chapter 20.3; we consider only the organization of a single module here.

## 9.2 Communication Between Functions

Before beginning this section, you should review the overview of functions and module organization that is given in Chapter 5.

The arrows show the ways that information flows to and from functions in Figure 9.3.



**Figure 9.1. Information flow in `pow2()`.**

One function in every program must be named `main()`; it is often called the *main program*. The **main program** can call other functions and those functions can call each other, but none can call `main()`. We use the term **caller** to refer to the function that makes the call and **subprogram** to refer to the called function. For example, in Figures 5.24 and 5.25, `main()` is the caller and `cyl_vol()` is a subprogram that is called from the second box in `main()`.

### 9.2.1 The Function Interface

Each function has a defined **interface**, which is declared by writing a prototype for the function. The prototype lists the return type and the type and name of one or more parameters. Either the parameter list or the return type may be replaced by `void`.

Each parameter listed in the prototype becomes a separate communication path by which the caller can send information into the function. The function's result or return value (if not `void`) is a communication path from the subprogram back to the caller, as is each address parameter. Information also can be passed between a caller and a subprogram through global variables, which will be discussed in Section 9.4. However, this practice is discouraged and should be avoided wherever possible. Extensive use of global variables can make a program difficult to debug because it vastly increases the number of possible interactions among program modules and introduces the possibility for unintended and undocumented interactions. Thus, modern programming style dictates that all information passed between caller and subprogram should go through the declared interface.

In Figure 9.1, the body of each function is shaded, indicating that it may appear as a “black box” to the programmer. The inner workings of a library function frequently are hidden from the programmer, like those of `exp()` in this case. You need not know the details of what is inside a function to be able to use it.

In contrast, interfaces are white. This symbolizes that the programmer can (and must) know the details of the interface. This information is supplied in a header file and by the documentation that normally accompanies a software library. **Header files** are used to keep the declarations consistent from module to module and to permit functions in one module to call functions in another.

The passage of information into the subprogram is represented by right-facing arrows. Function execution begins at the entry point and, when complete, control returns to the caller along a left-facing arrow, which ends at the return address in the caller. The caller continues processing from that point. The function also may return a result along the left-facing arrow.

A **function call** consists of the name of the function followed by a list of arguments in parentheses. Some functions have no parameters, in which case the parentheses in the function call still must be written but with nothing between them. During the calling process, two kinds of information are sent from the caller into the subprogram:

- One argument value for each declared parameter.
- The **return address**, which is the address of the first instruction in the *caller* after the function call. This address is passed by the caller to the subprogram on every call so that the function knows where to go when its execution is finished.

During a function call, the C run-time system allocates a stack frame, stores the argument values there, and stores the return address in the adjacent locations. Control is then transferred to the function, which allocates (and possibly initializes) storage for local variables. Control then jumps to the **entry point** of the function, which is the first line of code in its body. Execution of the subprogram begins and continues until the last

line of code is completed, the program aborts by calling `exit()`, or control reaches a `return` statement, which returns control to the caller at the return address, taking along any return value produced.

The `return type` declared in the prototype is the type of value that will be returned. If the `return` statement returns an answer of some other type, C will convert it to the declared type and return the converted value. If such a conversion is not possible, the compiler will issue an error comment. This is discussed more fully in Section 9.3.

**Missing prototypes.** The general rule in C is that everything must be declared before it is used. There are two ways to “declare” a function: either supply a prototype or give the complete function definition. To guarantee correctness, one or the other must occur in your program before any call on that function. The C compiler<sup>1</sup> must know the prototype of a function to check whether a call is legal. Sometimes, however, a programmer forgets to either `#include` a necessary header file or write a prototype for a locally defined function. Sometimes the prototype is in the file but in the wrong place, coming after the first call on the function.

In any of these cases, the compiler does *not* just give an error comment about a missing prototype and quit. The first time it encounters a call on a nonprototyped function it simply *makes up* a prototype and continues compiling. The compiler will use the types of the actual arguments in the call to construct a prototype that matches, but all such created prototypes have the return type `int`. Sometimes this prototype is exactly what the programmer intended; other times it is wrong because the call depends on an automatic type conversion or contains an error. In any case, the constructed prototype becomes the official prototype for the function and is used throughout the rest of the program. If it has an incorrect parameter or return type, the compiler will compile too many or too few type coercions for each function call.

If a misplaced prototype is found later in the file and it is the same as the prototype constructed by the compiler, there is no problem. However, if it is different, the compiler will give an error comment such as *type conflict in function declaration* or *illegal redefinition of function type*. This can be an astonishing error comment if the programmer does not realize that the problem is *where* the prototype was written, not *what* was in it. If the prototype really is missing, not just misplaced, a similar error comment may be produced when the compiler reaches the actual function definition. If you see such an error comment, check that all functions have correct prototypes and that they are at the top of the program.

### 9.2.2 Parameter Passing

When each function is called, a new and separate memory area, called a *stack frame*, is allocated for it. The function’s parameters are allocated in its stack frame and argument values are copied into the parameter during the function call. A function’s local variables are also in the frame, as is the information necessary to return from the function. Figure 9.2 is a diagram of the run time memory allocated for the program in Figure 5.24. As each function is called, a new frame for it is placed on the stack. First `main()` was called, then it called `cyl_vol()`, which called `pow()`. Parts of the `pow()` frame are gray because we have no idea how this library function is implemented or what its parameters are named.

When a function returns, its stack frame is deleted. Later a frame for another function might be stored in the same addresses. In the cylinder program, storage for the `cyl_surf()` function will end up in the memory locations that `cyl_vol()` had previously occupied, and the stack frame for `pow()` will be at a slightly different address because the frame for `cyl_surf()` is bigger than the frame for `cyl_vol()`.

**Call by value.** Most arguments in C are passed from the caller to the subprogram **by value**. This means that a copy of the value of the argument is sent into the subprogram and becomes the value of the corresponding parameter. The function does not know the address of the argument, which could be a variable or the result of an expression. If the argument is a variable, the subprogram cannot change the value stored there. For example, in Figure 5.25, the subprogram `cyl_vol()` receives the value of `main()`’s variable `diam` but not the address of `diam`. The code in the body of `cyl_vol()` can change the value of its own parameter, `d`, but doing so will not change the value stored in `main()`’s `diam`. Information cannot be passed back to the caller through an ordinary parameter.

---

<sup>1</sup>ISO C and older C implementations differ extensively on the rules for function prototypes, definitions, and type checking. In this text, we speak only of standard ISO C.

We see the memory allocated for the program in Figure 5.24 at two moments during run time. The diagram on the left shows memory just after the `pow()` function is called by `cyl_vol()`. The diagram on the right is a later moment, during execution of `cyl_surf()`.

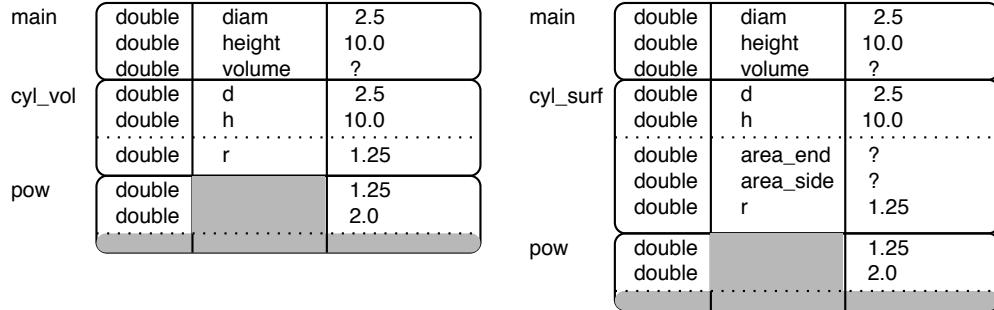


Figure 9.2. The run-time stack.

**Address parameters.** In contrast, some arguments are passed by passing the address of a variable (rather than its value) into the subprogram. The subprogram can both use and change the information at the argument address. An example of a function that sometimes must return more than one piece of information is `scanf()`. It uses the return value to return an error code, which we have ignored so far,<sup>2</sup> and it returns one or more data items through **address arguments**. When we call `scanf()`, we send it the address of each variable to be read. It reads the data, stores the input(s) in the given address(es), and returns a success or failure code. A programmer also can define such functions with address parameters; we explain how in Chapter 11.

### 9.3 Parameter Type Checking

The compiler uses a prototype for two purposes: checking whether the call is legal and compiling any type conversions necessary to make the argument types match the parameter types.

**Number of arguments.** If the number of arguments in a function call is appropriate, the compiler considers each parameter–argument pair, one by one, comparing the parameter type declared in the prototype to the type of the argument expression. If they match exactly, code is compiled to copy the argument values into the subprogram’s parameters and transfer control to its entry point. If the number of arguments supplied by a function call does not match the number declared in the prototype, the compiler prints an error comment<sup>3</sup>.

**Type coercion of mismatched arguments and parameters.** If the number of arguments is the same as the number of parameters but their **types do not match** exactly, the compiler will attempt to convert each argument to the declared parameter type according to the standard type-conversion rules. We already discussed a large number of variations of the basic integer, floating-point, and character types: `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `char`, `signed char`, `unsigned char`, `float`, `double`, and `long double`. Taken as a set, these are called the **arithmetic types**. An argument of any arithmetic type can be coerced to any other arithmetic type, if needed, to make the argument’s type match the type declared in the function’s prototype. An instance of such **type coercion**, is shown in the Figure 9.3; the argument in the call on `pow2()` is an `int`, while the parameter is a `double`. The C compiler will include code to convert the `int` argument value to type `double` as part of the function call, and the function will receive the `double` value that it expects.

**Meaningful conversions.** Type conversion or coercion is possible if there is a meaningful relationship between the two types, such as both being numbers. If conversion is not possible, the compiler will issue a

<sup>2</sup>This will be explained in Chapter 14.

<sup>3</sup>Some functions, such as `printf()` and `scanf()` permit the number of arguments to vary. Similar functions can be written using special argument-handling mechanisms supported by the `stdard` library.

---

```
#include <stdio.h>
#include <math.h>
int pow2 ( double x ) { return pow( 2.0, x ); }      // 2 to the power x

int main( void )
{
    for (int m=0; m<10; ++m) printf( "%10i\n", pow2(m) );
    return 0;
}
```

This program prints a table of the powers of 2 using a `double`→`int` function named `pow2()`. When `pow2()` is called from `main()` with an integer argument, the argument will be coerced to type `double` to match `pow2()`'s prototype. Within the function, the result of calling `pow()` will be coerced to type `int`, to match `pow2()`'s prototype, before being returned to `main()`.

---

**Figure 9.3.** The declared prototype controls all.

fatal error comment.<sup>4</sup> The rule in C is that any numeric type can be converted to any other numeric type. Therefore, a `short int` can be converted to a `long int` or an `unsigned int` or a `float` and vice versa. Some kinds of argument coercions are very common and compilers simply include code for the conversion and do not notify the programmer that it was necessary. For example, normally no warning would be given when a `float` value is coerced to type `double`. At other times, compilers warn the programmer that a conversion is occurring. This happens when an unusual kind of argument conversion would be required or the conversion might result in a loss of information due to a shortening of the representation, as when a `double` value is converted to type `float`. The warning you get depends on the nature of the type mismatch, the severity of the possible consequences, and your particular compiler.

**Type coercion of returned values.** The declared return type also is compared to the actual type of the value in the `return` statement. If they are different, the value will be coerced to the declared type before it is returned. The compiler generates the conversion code automatically. For example, in Figure 9.3, the value calculated by the expression in the `return` statement is of type `double` (the math-library functions always return `doubles`). However, the prototype for `pow2()` says that it returns an `int`. What happens? The C compiler will notice the type mismatch and compile code to coerce the `double` value to an `int` during the return process. The compiler may also give a warning message.

**Parameter order.** The order of the arguments in a function call is important. If a function's parameters are defined in one order and arguments are given in a different order in a call, the results generally will be nonsense. There is no "right" order for the parameters in a function definition; this is a design issue. However, once the definition has been written, the order of the arguments in the call must be the same. If the program has several functions that work with the same parameters, the designer should choose some order that makes sense and consistently stick to that order when defining the functions to avoid absentmindedly writing function calls with the arguments in the wrong order.

Sometimes a compiler, by performing its normal parameter type checking, can detect an error when a programmer scrambles the arguments in a function call. More often, this is not possible. For example, the function `cyl_vol()` in Figure 5.24 has two parameters, the diameter and height of a cylinder. Since the parameters are the same type, the compiler cannot tell when the programmer writes them in the wrong order. The result will be a program that compiles, runs, and produces the wrong answer. To further demonstrate the kind of nonsense that can result from mixed-up arguments, we will use a silly two-parameter function named `n_marks()` that inputs a number  $N$  and a character  $C$  and prints  $N$  copies of  $C$ . (Figure 9.4)

The output is

```
Enter a number and a character: 45.7 !
You entered 45.70 and !.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Compare this output to the output from the erroneous call in the next paragraph.

---

<sup>4</sup>If the function has an ANSI prototype, the coercions allowed for arguments and return values are the same as those allowed for assignment statements.

---

This program is used to show the effects of calling a function with parameters in the wrong order.

```
#include <stdio.h>
#include <math.h>
void n_marks( double n, char ch ); // function prototype

int main( void )
{
    char ch;      // The character to print
    double x;     // How many characters to print
    printf( "\nEnter a number and a character: " );
    scanf( "%lg %c", &x, &ch );
    printf( "\nYou entered %.2f and %c.", x, ch );
    n_marks( x, ch );           // Call function to print x many ch's
    return 0;
}
// -----
void n_marks( double n, char ch ) // Print n copies of character ch.
{
    putchar( '\n' );
    for (int k = 0; k<n; ++k) putchar( ch ); // print n characters
    printf( "\n\n" );                      // flush output to screen
}
```

---

**Figure 9.4.** The importance of parameter order.

**Parameter coercion errors.** When the compiler translates a function call, it either accepts the arguments as given, coerces them to the declared type of the parameter, or issues a fatal error comment. This automatic conversion can result in some weird and invalid output. For example, suppose a programmer called the `n_marks()` function but wrote the arguments in the wrong order: `n_marks( ch, x )`; The output would be:

```
Enter a number and a character: 45.7 !
You entered 45.70 and !.
-----
```

This call certainly is not what a programmer would intend to write. However, according to the ISO C standard, this is a legal call. The standard dictates that the character `ch` will be converted first to an `int` and then to a `double`, while the `double x` will be converted first to an `int` and then to a `char` to match the parameter types in the prototype `void n_marks( double, char )`. Using this input, the programmer would expect to see a line of 46 ! signs, but instead a line of 33 - signs is printed because of the conversions: The ASCII code for ! is 33, which will be converted to 33.00, and the 45.7 will be converted to 45, which is the code for -. Some compilers at least may give a warning comment about these two “suspicious” type conversions, but all standard ISO C compilers will compile the conversion code and produce an executable program. When you try to run such a program, the results will be nonsense, as shown.

## 9.4 Data Modularity

A large program may contain hundreds or thousands of objects (functions, variables, constants, types, etc.). If a programmer had to remember the name, purpose, and status of this many objects, large programs would be very hard to write and harder to debug. Happily, C supports modular programming. Each module has its own independent set of objects and interaction between modules can be strictly controlled. The same name can be

used twice, in different modules, to refer to different objects, so a programmer need not keep a mental catalog of the hundreds of names that might have been used. C's accessibility and visibility rules determine *where* a variable or constant may be used and *which* object a name denotes in each context. We use the program in Figures 9.5 and 9.7 to illustrate these concepts and the related concept of scope.

#### 9.4.1 Scope and Visibility

The **scope** of a name is the portion of the program in which it exists and can be used. The three levels of scope are local, global, and external; respectively meaning that access to an object can be restricted to a single program block or function, shared by all functions in the same file or program module, or shared by parts of the program in different files or program modules. More specifically,

- Parameters and variables defined within a function are completely **local**; no other functions have access

---

This program calculates the pressure of a tank of CO gas using two gas models. The functions in Figure 9.7 are part of this program and should be in the same source file. A call graph is given in Figure 9.6

```
#include <stdio.h>
#define R 8314           // universal gas constant
float ideal( float v );      // prototypes
float vander( float v );
float temp;                  // GLOBAL variable: not inside any function
                            // temperature of CO gas

int main( void )
{
    // Local Variables -----
    float m;                // mass of CO gas, in kilograms
    float vol;               // tank volume, in cubic meters
    float vmol;              // molar specific volume
    float pres;               // pressure (to be calculated)

    printf( "\n Input the temperature of CO gas (degrees K): " );
    scanf( "%g", &temp );
    printf( "\n The mass of the gas (kg) is: " );
    scanf( "%g", &m );
    printf( "\n The tank volume (cubic m) is: " );
    scanf( "%g", &vol );
    vmol = 28.011 * vol / m; // molar volume of CO gas
    pres = ideal( vmol );     // pressure; ideal gas model
    printf( "\n The ideal gas at %.3g K has pressure "
           "%.3f kPa \n", temp, pres );
    pres = vander( vmol );    // pressure; Van der Waal's model
    printf( "\n Van der Waal's gas has pressure "
           "%.3f kPa \n\n", pres );

    return 0;
}
```

---

Figure 9.5. Gas models before global elimination.

This is a function call graph for the gas pressure program in Figures 9.5 and 9.7.

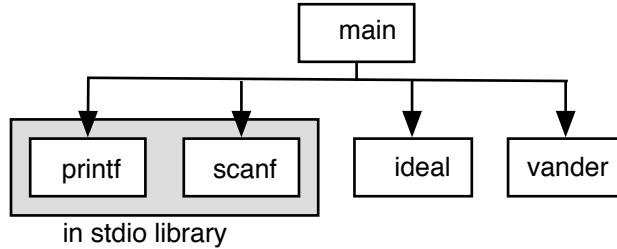


Figure 9.6. A call graph for the CO gas program.

to them.

- We are permitted to declare variables outside of the function definitions, either at the top of a file or between functions. These are called **global** variables. They can be used by other modules and all the functions in the code module that occur after their definition in the file. It is possible (but not the default) to restrict the use of these symbols to the module in which they are defined.
- All global names and all the functions defined in a module default to **extern**; that is, they are **external** symbols unless declared otherwise. This means that their names are given to the system's linker and all the modules linked together in a complete program can use these variables.<sup>5</sup>

<sup>5</sup>External linkage will be discussed in Chapters 19 and 23.

These functions are part of the gas models program in Figure 9.5 and are found at the bottom of the same source code file.

```

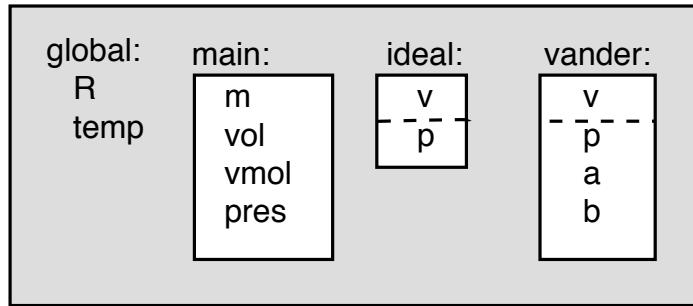
// -----
// Pressure of CO gas in a tank, using the ideal gas equation Pv = RT.
float ideal( float v )
{
    float p;                      // LOCAL variable DECLARATION
    p = R * temp / v;             // pressure in Pascals
    return p / 1000.0;            // pressure in kilo Pascals (kPa)
}
  
```

```

// -----
// Pressure of CO gas in a tank, using Van der Waal's equation,
// P = RT/(v-b) - a/(v*v).
float vander( float v )
{
    float p;                      // LOCAL declaration, not same p as above
    const float a = 1.474E+05;      // constants for CO gas
    const float b = .0395;
    p = R * temp / (v - b) - a / (v * v);   // pressure in Pascals
    return p / 1000.0;              // kPa pressure
}
  
```

Figure 9.7. Functions for gas models before global elimination.

The diagram shows the scope of the functions, variables, and constants defined in the gas models program from Figures 9.5 and 9.7. Each box represents one scope; the gray box represents the global scope. Local scopes are white boxes; within each, parameters are listed above the dotted line and local variables and constants below it. Symbols defined in the gray area are visible everywhere in the program where they are not masked. Symbols in the white boxes are visible only within the scope that defines them.



**Figure 9.8. Name scoping in gas models program, before global elimination.**

We say that a locally declared variable or parameter is **visible**, or **accessible**, from the point of its declaration until the block's closing `}`. This means that the statements within the block can use the declared name, but no other statements can (i.e., the scope and visibility of a local variable are the same). A global or external variable is visible everywhere in the program after its definition, *except* where another, locally declared variable bears the same name. If a function has a parameter or local variable named `x` and a statement `x = x+1`, the local `x` will be used no matter how many other objects named `x` are in the program's "universe." Therefore, the visibility of a global variable is that portion of its scope in which it is not masked or hidden by a local variable. This is a very important feature; it is what frees us from concern about conflicts with the names of all the hundreds of other objects in the program.

#### 9.4.2 Global and Local Names

Insofar as possible, all variables should be declared locally in the function that uses them. Information used by two or more functions should be passed via parameters. The use of global variables usually is a bad idea; any variable that can be seen and used by all the functions in a program might be changed erroneously by any part of the program. When global variables are in use, no one part of the program can be fully understood or debugged without considering the entire thing.

While global variables are not encouraged, constants and types usually are declared at the top of a file because they are necessary to coordinate the actions of different parts of a program. Since the values of `const` variables and `#defined` symbols cannot be changed, their global visibility does not foster the same kind of problems as a global variable. Further, declaring constants and types at the top of a file makes them easier to locate and revise, if necessary.

A function prototype can be declared globally or locally in every function that calls it. Each style of organization has its advantages. In this text, we declare most prototypes globally because it is simpler. We illustrate some of these issues using the program in Figure 9.5, which has two subprograms (Figure 9.7) and a variety of local and global declarations (Figure 9.8).

**Notes on Figures 9.5, 9.7, and 9.8. Gas models before global elimination.** We use a main program, two functions, a global constant, and a global variable to examine the scope and visibility of names in C. Figure 9.8 illustrates the scope of the symbols defined in this program.

***First box, Figure 9.5: global declarations and included files .***

- The `#include <stdio.h>` means that everything in the file `stdio.h` will be copied into this program at this point. All the objects declared in `stdio.h`, therefore, will have a global scope in this program.
- The constant `R` and the variable `temp` are declared globally. In Figure 9.8, these names are written in the gray box, which represents the global scope. Here, they are visible and can be used by any function in this

file; that is, by `main()`, `ideal()`, and `vander()`. The functions `ideal()` and `vander()` also can be called from any part of this file.

- It is considered very bad style to use a global variable such as `temp`. We do so only to demonstrate the meaning of global declarations and show how to eliminate them.
- A global constant such as `R` creates fewer problems than a global variable. It is common for a program to use global constants and is not considered bad style.

*Second box, Figure 9.5: declarations for main().*

- The four variables `m`, `vol`, `vmol`, and `pres` are local to `main()` and therefore visible only within `main()`. In Figure 9.8, the leftmost white rectangle represents the scope created by `main()`. Inside it is a list of `main()`'s local variables.
- The functions `ideal()` and `vander()` cannot access the values in `m`, `vol`, `vmol`, and `pres` because the values are local within `main()`.

*Third box, Figure 9.5: code for main().*

- This code can use the definitions and the global variable and constant defined in the first box as well as the variables defined in the second box.
- This code cannot use the parameters, variables, or constants defined by the two functions in Figure 9.7. If we tried to use `v`, `p`, `a`, or `b` here, it would cause an undefined-symbol error at compile time.
- The two inner boxes contain the calls on the functions `ideal()` and `vander()`. These functions will need to be modified to eliminate the use of the global variable.

*Sample output from this program is:*

```
Input the temperature of CO gas (degrees K): 28.5
```

```
The mass of the gas (kg) is: 1.2
```

```
The tank volume (cubic m) is: 3
```

```
The ideal gas at 28.5 K has pressure 3.385 kPa
```

```
Van der Waal's gas has pressure 3.357 kPa
```

*First box, Figure 9.7: code for the function ideal().*

- Because this function is compiled at the same time as the code in Figure 9.5, it can use any object, such as `R` or `temp`, that is defined (or included) in the first box in Figure 9.5.
- In Figure 9.8, the center rectangle represents the scope created by `ideal()`. Inside it are `ideal()`'s parameter `v` and the local variable `p`, which are visible only within `ideal()` and cannot be used outside this function.

*Second box, Figure 9.7: code for the function vander().*

- This function also can use objects such as `R` and `temp` that are defined (or included) in the first box in Figure 9.5.
- In Figure 9.8, the rightmost rectangle represents the scope created by `vander()`. Inside it are `vander()`'s parameter `v` and the local variables `a`, `b`, and `p` (`a` and `b` actually are constants). These are visible only within `vander()` and cannot be used outside this function.
- Each parameter or local variable declaration creates a new object. Therefore, the `p` defined here is a different variable than the `p` defined in `ideal()`, with its own memory location, even though they have the same name.
- If the local variable `p` had been named `temp`, this function would compile but not work properly because the local variable would mask the global variable. Every reference to `temp` in `vander()` would access the local variable, and the information in the global `temp` would not be accessible within the function. The next section shows how to improve the lines of communication so that these difficulties do not occur.

### 9.4.3 Eliminating Global Variables

We introduced a programming style in which interaction with the user is done by one function (often `main()`) and calculations by another. This separation of work makes a program maximally flexible and easier to modify at a later date. However, since one function reads the input data and another uses it for calculations, the data value must be communicated from the first to the second.

A beginning programmer often will be tempted to use a global variable because it provides one way to solve this communication problem. A global variable is visible to both the data input function and the calculation function, so nothing special has to be done to communicate the value from the first to the second.

In a small program, using global variables might seem easy and communicating through parameters might seem to be a nuisance. However, global variables almost always are a mistake,<sup>6</sup> because they allow unintended interactions between distant parts of the program. They make it hard to follow the flow of data through the process, and they make it harder to modify and extend the program. In a large program, global variables

---

<sup>6</sup>The program in Figure ?? shows an instance where the use of global variables is acceptable.

---

This program is a revised and improved version of the gas models program in Figures 9.5 and 9.7; it solves the communication problem by using a parameter instead of a global variable. The functions in Figure 9.10 are part of the revised program and should be in the same source file.

```
#include <stdio.h>
#define R 8314           // universal gas constant

float ideal( float v, float temp );
float vander( float v, float temp );

int main( void )
{
    // Local Variables -----
    float temp;          // Temperature of CO gas
    float m;              // mass of CO gas, in kilograms
    float vol;             // tank volume, in cubic meters
    float vmol;            // molar specific volume
    float pres;             // pressure (to be calculated)

    printf( "\n Input the temperature of CO gas (degrees K): " );
    scanf( "%g", &temp );
    printf( "\n The mass of the gas (kg) is: " );
    scanf( "%g", &m );
    printf( "\n The tank volume (cubic m) is: " );
    scanf( "%g", &vol );
    vmol = 28.011 * vol / m;           // molar volume of CO gas
    pres = ideal( vmol, temp );      // pressure; ideal gas model
    printf( "\n The ideal gas at %.3g K has pressure "
           "%.3f kPa \n", temp, pres );
    pres = vander( vmol, temp );     // pressure; Van der Waal's model
    printf( " Van der Waal's gas has pressure "
           "%.3f kPa \n\n", pres );
    return 0;
}
```

---

Figure 9.9. Eliminating the global variable.

---

These functions illustrate how to eliminate a global variable from Figure 9.7. The boxes highlight the changes necessary to replace the global variable by adding a parameter to each function.

```

// -----
// Pressure of CO gas in a tank, using the ideal gas equation Pv = RT.
*/
float ideal( float v, float temp )
{
    float p;                      // LOCAL variable DECLARATION
    p = R * temp / v;             // pressure in Pascals
    return p / 1000.0;            // pressure in kilo Pascals (kPa)
}

// -----
// Pressure of CO gas in a tank, using Van der Waal's equation,
// P = RT/(v-b) - a/(v*v).
*/
float vander( float v, float temp )
{
    float p;                      // LOCAL declaration, not same p as above
    const float a = 1.474E+05;
    const float b = .0395;          // constants for CO gas
    p = R * temp / (v - b) - a / (v * v); // pressure in Pascals
    return p / 1000.0;            // kPa pressure
}

```

---

**Figure 9.10. Functions for gas models after global elimination.**

become a debugging nightmare. It is hard to know what parts of the program change them and under what conditions. Therefore, it is important, from the beginning, to avoid global variables and learn to use parameters effectively.

We will use the program in Figures 9.5 and 9.7 to demonstrate the technique for eliminating global variables. The result is shown in Figures 9.9 and 9.10. Parameters are the right way to solve the communication problem. They make the sharing of data explicit and they prevent unintended sharing with unrelated functions. In general, global variables should be replaced by parameters. The transformation works for globals used to send information into a function; a variant of this technique<sup>7</sup> is needed if the function also uses the global variables to send information back out.

**Notes on Figures 9.9 and 9.10. Eliminating the global variable.** We start with the main program in Figure 9.5 and the two functions in Figure 9.7, which communicate through a global variable. To eliminate this variable and replace it by a parameter, we need to change five lines in Figure 9.5 and two lines in Figure 9.7.

**First box of Figure 9.9.** The prototypes for the functions `ideal()` and `vander()` found in the first box of Figure 9.5 need to be changed to include an additional parameter of the same type as the global variable. In both cases, we add the new parameter `second`. It is important to be consistent about parameter order when adding parameters, to avoid the problems mentioned earlier.

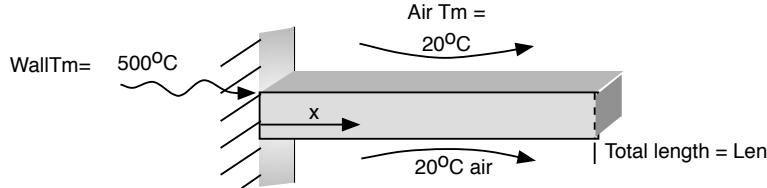
**Second box of Figure 9.9.** The declaration of `temp` must be moved out of the global area where it was defined in Figure 9.5 and into `main()`'s local area.

**Third and fourth boxes of Figure 9.9.** The function calls in these boxes must be modified to include an additional argument, the value of the former global variable, which is now a local variable in `main()`.

---

<sup>7</sup>This variation uses pointer parameters, which will be covered in Chapter 11.

**Problem scope:** A long, slender cooling fin of rectangular cross section extends out from a wall, as shown. Print a table of temperatures every 0.005 m along the fin.



**Formulas:** The temperature of the fin at a distance  $x$  from the wall is:

$$\text{Temperature}(x) = \text{AirTm} + (\text{WallTm} - \text{AirTm}) \times \frac{\cosh[\text{FinC} \times (\text{Len} - x)]}{\cosh(\text{FinC} \times \text{Length})}$$

#### Constants:

Temperature of the air,  $\text{AirTm} = 20^\circ\text{C}$

Temperature of the wall at the base of the fin,  $\text{WallTm} = 500^\circ\text{C}$

Fin constant,  $\text{FinC} = 26.5$

**Input:** Length of the fin,  $\text{Len}$ , in meters, which must be greater than zero.

**Output required:** Print column headings **Distance from base (m)** and **Temperature (C)**. Beneath the headings print a table of temperatures starting with the temperature at the wall ( $x = 0$ ). Print one line for each increment of 0.005 m up to the length of the fin.

**Computational requirements:** Print the distance from the wall to three decimal places and the temperature to one.

**Test plan:** Using the numbers from the diagram, for a wall that is .6 meters long, the temperature at the wall should be  $500^\circ\text{C}$  and .01m from the wall, the temperature should be  $398.535^\circ\text{C}$ .

**Figure 9.11. Problem specification and test plan: fin temperature.**

**First box of Figure 9.10.** A new parameter was added to the parameter list in the prototype for `ideal()`. We also must add the parameter to the function header.

**Second box of Figure 9.10.** We must add the new parameter to the list for `vander()` as well.

## 9.5 Program Design and Construction

In the gas pressure example, we started with a program and its two functions and analyzed it section by section. This is a good way to understand how a given program works, but it gives little perspective on how a program with functions is developed. In this section, we reverse the process and show how to develop a program and its subprograms from the specification. Section 9.5.1 lists the steps and describes them briefly. Section 9.5.2 explains these steps more fully and applies them to a real problem.

### 9.5.1 The Process

**The DNA: a complete specification.** The first step in designing a program is to decide what you want the program to do and specify it as precisely as possible. If there is any doubt about the specifications or any missing information, this must be cleared up before proceeding.

**Start with the “skin” of `main()`.** Write the routine portions of `main()`. If you wish to test or run the program on several data sets, write a work loop in the body of `main()`.

**Define the skeleton of the work to be done.** Start by listing the major phases in processing a single data set. Generally these phases are input, calculation, and output; but one of these phases might not be needed and the calculation phase may have multiple steps. Write the code to perform each phase if it is only one or two lines long. Otherwise, invent a name for a function that will do each task. These statements will be in a `work()` function if you have one, otherwise in `main()`.

**The circulatory system: declarations and prototypes.** Go back to the top of `main()` and write whatever declarations and prototypes you will need to support your skeleton code. These do not have to be perfect. Write a comment for each one.

**Health check: compile.** It is much easier to find compiler errors when you compile and check the code a little bit at a time. However, a program will not compile if it calls functions that have not yet been written.

One solution to this is to write a *stub* for each function that has been named but not yet defined. A stub is a function that does nothing except print its own greeting message and return some value of the correct type (or `void`). The return value does not need to be meaningful; any definition is OK, as long as its parameter types and return value type match the prototype. Compile the program initially as soon as the stubs are written. Then later, after every function is fleshed-out, compile the program again.

An alternative to writing function stubs is to temporarily **amputate** calls on functions that have not yet been written by enclosing them in comment marks. Sometimes entire lines are amputated. At other times, a function call is commented out and replaced by a literal constant. The comments can easily be removed when the function is written, later.

**The brains: developing the functions.** Tackle the functions one at a time, in any convenient order. For each, write a comment block that describes the purpose of the function and any preconditions. Then go through the same steps as for `main()`: routine parts, skeleton, declarations, and possibly more prototypes and stubs. As you learn more about your functions, you may need to add parameters to the prototypes you have previously written.

**Integration and testing.** Combine and test all of the program parts according to your plan.

### 9.5.2 Applying the Process: Stepwise Development of a Program

We now apply the design steps from Section 9.5.1 to develop a program for a real problem: calculating the temperature of a cooling fin. This problem arose from a real application: designing the cooling apparatus for a piece of machinery. Figure 9.11 is a diagram of the cooling fin and specifications for a program to analyze its temperature gradient. The problem specification comes directly from the set of engineering principles and formulas in use by the designer.<sup>8</sup>

A fin is slender if its length is an order of magnitude greater than the height or width of its rectangular cross section. The cooling properties of a fin depend on temperatures of the wall and the air and a fin constant,  $FinC$ , which is a function of the film coefficient of the air and the thermal conductivity and cross-sectional area of the fin.

#### Steps in writing `main()`.

- **The skin.** We start by writing the `#include` commands (`stdio`, as usual, and `math` because this is a numeric application.) and the first and last few lines of `main()`, the greeting message and return statement

```
#include <stdio.h>
#include <math.h>
int main( void )
{
    puts( " Temperature Along a Cooling Fin" );
    // Program code will go here.
    return 0;
}
```

---

<sup>8</sup>F. Incropera and D. DeWitt, *Fundamentals of Heat and Mass Transfer*, 4th ed. (New York: Wiley, 1996).

- *Multiple data sets.* We will write this program for only one data set, so we do not need a processing loop or a `work()` function. Therefore, the skeleton of the program becomes the body of `main()`.
- *The skeleton.* Next, prepare the **function skeleton**; We start by listing the two major phases in creating a table for a single fin situation. We need two basic steps:
  - a. Input and validate a fin length.
  - b. Print a temperature table for that length.

We write the code for step (a) directly because it requires only a few lines of code.

```
do {
    printf( " Please enter length of the fin (> 0.0 m): " );
    scanf( "%g", &Length );
} while (Length <= 0);
```

- *The circulatory system.* We finish step (a) by writing the declaration for `Length`.

```
float Length; // Length of the cooling fin.
```

- Step (b) is more complex, however, so we invent a function to perform the task and name it `print_table()`. We write a first draft of the prototype for `print_table()`, giving it the parameters and return type we think it will need. This information comes from the formula given in the specification: all values are constant except the fin length, so the length is the only parameter needed. We set the return type to `void` because most printing functions are `void`.

```
void print_table( float Length );
```

Only the prototype is written at this stage; the code itself will be written later. This is just a first guess at the proper prototype: if there are too many parameters or too few, or the types are wrong, that will be corrected later. Now we return our attention to `main()` and write a call on the new function.

```
print_table( Length );
```

If (unlike this example) the new function returns a value, we must store the result in a variable and we may need to write a declaration for that variable.

- *Health check.* The first draft of the main program is now complete; it is shown in Figure 9.12 with the corresponding call graph. We would like to use the C compiler to check the work so far but we cannot compile the program as it stands because the definition of `print_table()` is missing. So we create a stub for `print_table()`. This consists of an identifying comment, a function header that matches the prototype, and a single output statement to let the programmer track the program's progress. Since this function has a parameter, we use the stub to print its value, giving us confidence that the data is being correctly communicated to the function. We put this stub at the end of the program.

The stub served its purpose when we compiled this file. There were a few typographical errors, but the program is so short that they were easy to find and fix. The corrected code is shown in Figure 9.12. It ran successfully, producing this output:

```
Temperature Along a Cooling Fin
Please enter length of the fin (m): .06
>>> Entering print_table with parameter 0.06
```

From this, we can see that the program is starting and ending properly and receiving its input correctly.

- *The brains.* We now can start work on the function. If a program has two or more functions, we code them one at a time, in any convenient order. For each, we work on the skin, the skeleton, the circulation, and the brains.

Sometimes this leads to inventing another function; sometimes it means writing statements that compute the formulas given in the specification.

### Steps in writing print\_table().

- *The DNA and skin.* We have written a prototype and a function stub for `print_table()`. Now we need to think carefully about the function itself. One part of that process is to write a set of *preconditions* for the function, that is, things that must be true when the function is called. This function has one parameter, the length of the fin, which must (obviously) be a positive number, so we add a line to the comment block to document this precondition. This precondition announces that it is the job of the caller `main()` to validate the input – it will not be validated here. We now have this skeleton:

```
// Stub: -----
// Print a table of temperatures for a fin of given length, in meters.
// The length must be greater than 0.

void print_table( float Length )
{
    printf( " >>> Entering print_table with parameter %g", Length );
}
```

This code was written piece by piece on the preceding few pages. The corresponding call graph follows.

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

void print_table( float Length );
int main( void )
{
    float Length;      // Input: the length of the cooling fin.
    puts( " Temperature Along a Cooling Fin" );
    do {
        printf( " Please enter length of the fin (> 0.0 m): " );
        scanf( "%g", &Length );
    } while (Length <= 0);
    print_table( Length );
    return 0;
}

// Stub: -----
// Print a table of temperatures for a fin of given length, in meters.

void print_table( float Length )
{
    printf( " >>> Entering print_table with parameter %g", Length );
}
```

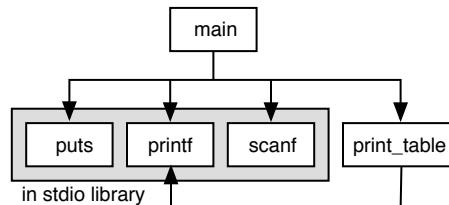


Figure 9.12. First draft of `main()` for the fin program, with a function stub.

- *The skeleton.*

To print a table, we must

1. print table headings,
2. print several lines of detail, and
3. print table footings.

Steps (1) and (3) are simple enough to write directly. We do that and leave space to insert step (2) later.

```
printf( "\n Distance from base (m)      Temperature (C) \n" );
printf( " -----      ----- \n" );
    // STEP 2 WILL GO HERE
printf( " -----      ----- \n" );
```

- *The circulatory system.*

We need a constant in the `print_table()` function: the step size for the loop. The specification says that the temperature should be calculated every 0.005 meters, but it always is unwise to bury constants like that in the code. We define the step size as a local constant so that it will be easy to modify in the future. It is obvious that we also need at least one variable, `distance`, to hold the current distance from the wall and another, `temp`, to hold the result of the temperature calculation:

```
float dist;           // Distance from wall.
float temp;          // Temperature at that distance.
const float step = .005; // Step size for loop.
```

For step 2, we need a loop that will produce one line of output on each iteration. For each line, we must compute and print the temperature at the current distance,  $x$ , from the wall. We invent a function named `compute_temp()` to compute the temperature. It needs a parameter,  $x$ , which we declare as type `float`, like all the other variables in this program. The function result is a temperature, so that will also be type `float`. The prototype becomes:

```
float compute_temp( float x );
```

The revised function call graph is shown in Figure 9.13.

- *The brains.*

Printing the detailed lines of the table requires a loop that starts at distance 0.0 from the wall and increases to the length of the fin in increments of the defined step size. Since the loop variable, `dist`, is not an integer, we need an epsilon value for the floating comparison that ends the loop. This epsilon must be smaller than the step size; we arbitrarily set it to half the step size:

```
float epsilon = step/2.0; // Tolerance
```

We are ready to code the loop. We start with the loop skeleton and defer the computation and printout:

```
for (dist = 0.0; dist < Length+epsilon; dist += step) { //Defer}
```

Now we approach the loop body. The loop prints the lines of the table one at a time. For each one, we must compute the current distance,  $x$ , from the wall and the temperature at distance  $x$ . The distance computation is handled by the loop control; the remaining tasks are done by `compute_temp()` and `printf()`:

```
temp = compute_temp( dist );
printf( "%12.3f %24.1f \n", dist, temp );
```

In the format, we use `%f` conversion specifiers because we want a neat table with the same number of decimal places printed for every line (3 for `dist` and 1 for `temp`, written with `%12.3f` and `%24.1f`, respectively). We supply a field width so that the output will appear in neat columns. To find the correct field width, we either count the letters in the headings or guess; a guess that is too big or too small can be adjusted after we see the output. We now have completed the first draft of the code for `print_table()`; it is shown in Figure 9.13.

```

// -----
// Print a table of temperatures for a fin "Length" meters long.
// Length must be greater than 0.

void print_table( float Length )
{
    float dist;           // Distance from wall.
    float temp;           // Temperature at that distance.
    const float step = .005; // Step size for loop.
    float epsilon = step/2.0; // Tolerance

    printf( "\n Distance from base (m)      Temperature (C) \n"
    printf( " -----      ----- \n" );
    for (dist = 0.0; dist < Length + epsilon; dist += step) {
        temp = 1.1; // compute_temp( dist );
        printf( "%12.3f %24.1f \n", dist, temp );
    }
    printf( " -----      ----- \n" );
}

```

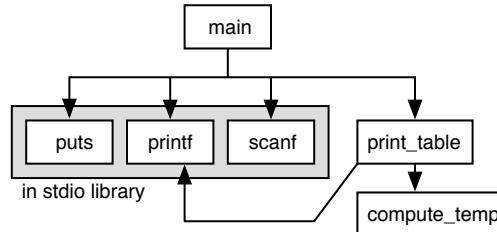


Figure 9.13. Fin program: First draft of `print_table()` function.

- *Health check.*

While coding the brains of a function, a programmer sometimes discovers that the function needs more information than its parameters supply. If that information is not supplied by global constants, the function header, prototype, and call(s) must be edited to supply the added data.

In the process of writing `print_table()`, we did not need to change the parameter list, so the call on `print_table()` inside `main()` still is correct and we do not need to modify `main()` at this time. (However, we might need to do this later.)

We have now completed a main program and one function with a tentative call on a second function that has not been written. It is time to compile again. We cannot compile the program as it stands because of the call on `compute_temp()`. Although we could write another function stub, in this case we choose to temporarily “amputate” the call on `compute_temp()` by enclosing it in a comment and setting the variable `temp` to an arbitrary (but recognizable) constant value:

```
temp = 1.1; // compute_temp( dist );
```

Now we compile the program, fix any compilation errors, and run it. Our program ran successfully, producing the output that follows (only the first and last few lines are shown).

Looking at this table, we see that the number of rows printed is correct and the numbers are adequately centered under the headings. The numbers in the first column are correct and the numbers in the second column are the constant value we used when we amputated the call on `compute_temp()`.

```
Temperature Along a Cooling Fin
Please enter length of the fin (m): .06
```

Distance from base (m)	Temperature (C)
0.000	1.1
0.005	1.1
0.010	1.1
...	...
0.050	1.1
0.055	1.1
0.060	1.1

### Steps in writing `compute_temp()`.

- *The DNA and the skin.*

The `compute_temp()` function must calculate the temperature of the fin at a given distance from the wall. Tentatively, we have given it one parameter, a `float` named `x`, which will range between 0 meters and the length of the fin. The function must return a `float` value to `print_table()`. We write a comment block and the shell of the function:

```
// -----
// Compute and return the temperature at distance x from the wall.
// x must be between 0.0 meters and the length of the fin.

float compute_temp( float x ){// Code goes here.}
```

- *The skeleton and the brains.*

We compute the temperature using the formula in the specifications (see Figure 9.11) and return the answer. This is a simple task that need not be broken down further. Here is the resulting statement:

```
return AirTm + (WallTm-AirTm) * cosh( FinCnow*(Length-x) ) / cosh( FinC*Length );
```

Looking at this formula, we see that it involves several variables (wall temperature, air temperature, fin constant, and length of the fin), not just the distance `x` from the wall. However, the first three values are given as constant numbers in the problem specification. We could write constants for these quantities in our formula, but the program would be much more useful if these values could be varied. Therefore, we choose to define all three constants in `main()` and add them to the parameter lists of both `print_table()` and `compute_temp()`. By placing all these constants in `main()`, we make them easy to find and modify. It also would be easy to change them into input variables if that were desired. Last, the fin length is a missing parameter; it is read in `main()` and we must pass it from `main()` through `print_table()` to this function. Similarly, we must pass the constants from `main()` to `compute_temp()`. To do so, we need to change the prototypes and headers of both functions and correct two function calls.

- *Circulation.*

We now have three constants and one input variable that must be passed from `main()` to `compute_temp()` as parameters. Although the order of the new parameters is not crucial, we want an order that makes sense and can be remembered, so we put all the properties of the fin together. Our new prototypes are

```
void print_table( float Length, float FinC, float WallTm, float AirTm );
float compute_temp( float x, float Length, float FinC, float WallTm, float AirTm )
```

Next, we add three constant definitions to `main()` and change the call on `print_table()` to use them:

```
const float FinC = 26.5;      // Fin constant.
const float WallTm = 500.0; // Wall temperature, C
const float AirTm = 20.0;    // Air temperature, C
print_table( Length, FinC, WallTm, AirTm );
```

Last, the function call in `print_table()` must also be changed:

```
temp = compute_temp( dist, Length, FinC, WallTm, AirTm );
```

- *Health check.* The completed program is shown in Figure 9.14. The first and last few lines of the output are shown below. The final development step is to compare these results to the ones in the test plan, to verify that we are fully finished.

```

Temperature Along a Cooling Fin
Please enter length of the fin (m): .06

Distance from base (m)      Temperature (C)
-----
0.000                      500.0
0.005                      445.5
0.010                      398.5
.....
0.055                      209.6
0.060                      208.0
-----
```

## 9.6 What You Should Remember

### 9.6.1 Major Concepts

**Modular design.** Large programs are organized into modules, which contain related sets of functions and constants. It is the compiler's job to properly compile and link together the files containing the modules. The organization of a module follows a stylistic pattern. Modular design is a skill worth learning if you intend to write programs of any substantial size.

**Parameter passing.** Most parameters are passed by value; that is, the value of the argument is copied into the parameter variable. This isolates the subprogram from the caller, making it impossible for the subprogram to change the values of the caller's variables. In Chapter 11, another parameter-passing mechanism (call by value/address) will be introduced that can support two-way communication between caller and subprogram.

**Parameter and return value type conversion.** The type of a function argument in a call should match the type of the corresponding formal parameter in the prototype. If they are not identical, the argument will be coerced (automatically converted) to the parameter's type, if that is possible. Any numeric type (including `char`) can be converted to any other numeric type. Similarly, the value returned by a function will be converted to the declared return type. Normally, this type coercion causes no trouble. However, converting from a longer representation to a shorter one can cause overflow or loss of precision and converting from `char` to anything except `int` usually is wrong.

**Parameter names.** The name of a parameter is arbitrary. It identifies the parameter value within the function and therefore should be meaningful, but it has no connection to anything outside the function, even if other objects have the same name. The order in which arguments are given in the call, not their names, determines which parameter receives each argument value.

**Parameter order.** The order of parameters for a function is arbitrary. However, related parameters should be grouped together, and when several functions are defined with similar parameters, the order should be consistent. The number and order of arguments are not arbitrary. Parameters and arguments are paired according to the order in which they are written (not by name or type).

**Call graphs.** A function call graph is a diagram that shows the caller-subprogram relationships of all the functions in a program. More sophisticated forms may include descriptions of the information being sent into and out of the subprogram.

**Scope and visibility.** The scope of an object is the portion of a program in which it exists. The visibility of an object is the portion of its scope in which it can be accessed. An object can have external, global, or local scope, meaning that it is available to the entire program, restricted to a single module, or restricted to a single function, respectively.

**Stub testing.** When a program is long and has many functions, stub testing often is the best way to construct and debug it. In this technique, the main program is written first, along with a stub for each function it calls. The stub is a function header, some comments, and only enough code to print the parameter values. If not a `void` function, it also must return some fixed and arbitrary answer. After the main function compiles and runs correctly by calling the stubs, the stubs are filled in, one at a time, with real code. This code, in turn, may

---

The problem specifications are given in Figure 9.11.

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

void print_table( float Length, float FinC, float WallTm, float AirTm );
float compute_temp( float x, float Length, float FinC, float WallTm, float AirTm );

int main( void )
{
    float Length;                      // Length of the cooling fin, in meters.
    const float FinC = 26.5;            // Fin constant.
    const float WallTm = 500.0;         // Wall temperature, C.
    const float AirTm = 20.0;           // Air temperature, C.

    puts( " Temperature Along a Cooling Fin" );
    do {
        printf( " Please enter length of the fin (> 0.0 m): " );
        scanf( "%g", &Length );
    } while (Length <= 0);
    print_table( Length, FinC, WallTm, AirTm );
    return 0;
}
// -----
// Print a table of temperatures for a fin that is "Length" meters long.
// Length must be greater than 0.

void
print_table( float Length, float FinC, float WallTm, float AirTm )
{
    float dist;                      // Distance from wall.
    float temp;                       // Temperature at x.
    const float step = .005;          // Step size for loop.
    const float epsilon = step/2.0;   // Tolerance.

    printf( "\n Distance from base (m)      Temperature (C) \n" );
    printf( " -----      ----- \n" );
    for (dist = 0.0; dist < Length + epsilon; dist += step) {
        temp = compute_temp( dist, Length, FinC, WallTm, AirTm );
        printf( "%12.3f %24.1f \n", dist, temp);
    }
    printf( " -----      ----- \n" );
}

// -----
// Compute the temperature at distance x from wall; 0<=x<=Length

float
compute_temp( float x, float Length, float FinC, float WallTm, float AirTm )
{
    return AirTm + (WallTm - AirTm) * cosh( FinC*(Length-x) ) / cosh( FinC*Length );
}
```

---

Figure 9.14. Temperature along a cooling fin.

require the construction of new stubs. After one or a few stubs have been fleshed out, the code is compiled and tested again. This is repeated until all stubs have been replaced by complete functions. This technique is important because it forces the programmer to work in a structured way; it ensures that all parts of the program are kept consistent with each other as they are developed. Also, compiler errors are easier to find and fix because there is never a large amount of new code being compiled for the first time.

### 9.6.2 Programming Style

**Monolithic vs. modular.** If you cannot look at a function on your computer screen and understand what it is doing, it is too long, too complex, or both. Keep function definitions short enough to see the beginning and the end at the same time. Generally, it is good to keep code for user interaction and code for mathematical computation in separate functions. Modular development also aids the compiling and debugging process.

**Placement of prototypes in the file.** The easiest way to be sure that the compiler uses the correct prototype to translate every function call is to write all the `#include` commands and all your prototypes at the top of each code module. Although other arrangements may be legal, according to C's rules, putting the prototypes at the top is the easiest way to avoid errors caused by misplaced prototypes, missing arguments, and incorrect argument order in function calls.

**Global vs. local.** Define variables locally, not globally, wherever possible. This tactic makes logical errors easier to locate and fix because it limits unintentional interaction between parts of a program. In general, parameters should be used for interfunction communication. Global definitions should be used only for new type definitions and constants shared by several parts of a program.

**Function documentation.** Every function should start with a highly visible comment line, such as a row of dashes. This line is very useful during debugging because it helps you find the functions quickly. You should be able to give a succinct description of the purpose of each function. This description should start on the second line of the comment at the top of the function definition. This comment also must make clear the purpose of each parameter and include a discussion of any preconditions.

**Parameter and argument consistency.** If more than one function uses the same set of parameters, reduce confusion by being consistent in their order and naming.

**Function layout.** The parts of a function definition can be arranged in many ways. The layout recommended here is the easiest to read and extends best to advanced programming in C++.

- Every function definition should start with a comment block, as described previously. The first line of code should be the return type and a comment, if needed, that describes the meaning of the return value (nothing else).
- The second line of code should start with the name of the function, followed by a left parenthesis. If all parameters will fit on one line, they should come next, ending with a right parenthesis. Otherwise, put each parameter on a separate line, with a comment. Put the closing right parenthesis on a line by itself, aligned directly under the matching left parenthesis.
- On the next line, write the left curly bracket in the first column.
- Next come the declarations with their comments; indent them two to four columns. Then leave a blank line and write the function body, indented similarly. Increase the indentation for each conditional or loop statement.
- Write the right curly bracket that ends the function in the leftmost column on a separate line.

### 9.6.3 Sticky Points and Common Errors

**Parameter order.** The arguments in a function call must be written in the same order as the matching parameters in the definition. If the order is scrambled, the code often will compile and run but produce incorrect results. Using an incorrect number of arguments can lead to confusing errors at compile time.

**The declaration must precede the call.** Either a function prototype or the complete function definition must precede all calls on the function. If this is not done, the compiler will construct a prototype automatically, which often will be wrong. This normally results in an error comment about an *illegal function redefinition* when the function definition is translated.

**Call vs. prototype vs. definition.** Beginning programmers often confuse the syntax of a function call with the syntax of the corresponding prototype and header. These have parallel but different forms. The function call supplies a list of argument values; although each value has a type, the type names are not written in the call. In contrast, the function prototype and header do not know what values eventually will be supplied by future calls, so they list the types of the parameters. In addition, the function header must give parameter names, so that the code can refer to the parameters and use them. Last, a semicolon is found at the end of a prototype but not at the end of a function header.

**A global vs. local mix-up.** Having global variables leads to trouble for a variety of reasons. One scenario is the following: A local variable declaration is omitted, and it happens to have the same name as a global variable. The compiler cannot detect the omission and will use the global variable. Any assignments to this variable will change the global value, causing unexpected side effects in other parts of the program. Such errors are hard to track down because they are not in the part of the program that seems to be wrong and produces incorrect results.

#### 9.6.4 New and Revisited Vocabulary

A large number of terms relating to functions and function calls have been introduced in this chapter. This list is provided as a review of the new concepts covered.

main program	function call	return type
subprogram	calling sequence	return value
function	caller	scope
module	call graph	visibility
library	argument	accessibility
header file	call by value	local
prototype	type coercion	global
function definition	arithmetic types	stack frame
interface	call by address	external
formal parameter	address argument	preconditions
parameter list	entry point	function skeleton
parameter order	return address	function stub
type matching	return statement	stub testing
type conflict	missing prototype	testing by amputation

## 9.7 Exercises

### 9.7.1 Self-Test Exercises

1. Call graph. The main program, which follows, calls two of the functions declared at the top. Which standard libraries would need to be included to compile this program? Draw a function call graph for this program.

```
#include <stdio.h>
#define MAX 7

void pattern( int p );
void stars( int n ) { for (int k = 0; k < n; ++k) printf( "*" ); }
void space( int m, int n ) { for (int k = 0; k <= n; k += 2) printf( " %i", m ); }
int odd( int n ) { return n % 2; }

int main( void ) //-----
{
    for (int k = 0; k < MAX; ++k) {
        if (k < 2 || k >= MAX-2) stars( MAX );
    }
}
```

```

        else pattern( k );
        printf( "\n" );
    }
}

void pattern( int p ) //-----
{
    stars( 1 );
    space( p, MAX-4 );
    if (odd(MAX)==1) printf( " " );
    stars( 1 );
}

```

2. Tracing calls. Trace the execution of the code in problem 2. Make a list of the function calls, one per line, listing the name, arguments, and return value for each. Show the program's output as well.
3. Local and Global. Look on the web site at the program for Newton's method. Fill in the following chart listing the symbols *defined* (not used) in each program scope. List the global symbols on the first line and add one line below that for each function in the program; remember that every function definition creates a new scope. The line for `main()` has been started for you.

Scope	Parameters	Variables	Constants
global	—	—	—
main()	—	—	—
	—	—	—
	—	—	—

4. Local names. List all the local symbols defined in the main program in Figure 9.4. List the parameters and local variables in the function named `n_marks()`.
5. Visibility. List all the variable names used in the function named `cyl_vol()` in Figure 5.24. For each, say whether it is a parameter or a local variable.
6. Control flow. Draw a flow diagram for the fin temperature program in Figure 9.14.
7. Prototypes and calls. Given the prototypes and declarations that follow, say whether each of the lettered function calls is legal and meaningful. If the call would cause a type conversion of either an argument or the return value, say so. If the call has an error, fix it.

```

void    squawk( int );
int     triple( int );
double   power( double, int );

int j, k;
float f;
double x, y;

(a) squawk( 3 );
(b) squawk( f );
(c) triple( 3 );
(d) f = triple( k );
(e) j = squawk( k );
(f) y = power( 3, x );
(g) y = power( x, 3 );
(h) x = power( double y, int k );
(i) y = power( triple( k ), x );
(j) printf( "%i %i", k, triple( k ) );

```

### 9.7.2 Using Pencil and Paper

1. Control flow. Draw a flow diagram for the Newton's method program on the text web site.
2. Call graph. Draw a call graph for the `n_marks` program in Figure 9.4.
3. Tracing calls. Trace the execution of the program in exercise 4, above. Make a list of the function calls, one per line, listing the name, arguments, and return value for each. Show the program's output as well.
4. Prototypes and calls. Given the prototypes and declarations that follow, say whether each of the lettered function calls is legal and meaningful. If the call would cause a type conversion of either an argument or a return value, say so. If the call has an error, fix it.

```

double rand_dub( void );
int half( double );
int series( int, int, double );

int j, k;
float f;
double x, y;

(a) half( 5 );
(b) rand_dub( y );
(c) x = rand_dub();
(d) j = half();
(e) f = half( x );
(f) j = series( x, 5 );
(g) j = series( 5, (int)x, y );
(h) y = series( j, k, rand_dub() );
(i) printf( "%g %g", x, half( x ) );
(j) printf( "%i %g", k, rand_dub( k ) );

```

5. Call graph. The main program that follows calls the three functions defined at the top. Which standard libraries would need to be included to compile this program? Draw a function call graph for this program.

```

#include <stdio.h>
#include <math.h>

double f( double x ) { return x / 2.0; }
double g( double x ) { return 1.0 + x; }
double h( double x ) { return x * 3.0; }

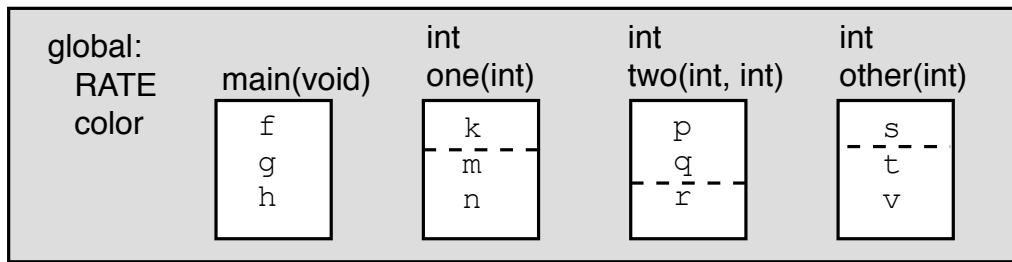
int main( void )
{
    double x = 1;
    double sum = 0.0;
    while (sum < 100) {
        x = h( x );
        printf( " %6.2f \t", x );
        sum += x;
        if (fmod( x, 2.0 ) == 0) x = f( x );
        else x = g( x );
        printf( " %6.2f \n", x );
    }
    printf( " ----- \n %6.2f \n", sum );
}

```

6. Local and Global. Look at the program for fin temperatures in Figure 9.14. Fill in the following table, listing the symbols that are *defined* (not used) in each program scope. List the global symbols on the first line and add one line below that for each function in the program (the line for `main()` has been started for you).

Scope	Parameters	Variables	Constants
global	—	—	—
main()	—	—	—

7. Visibility. The diagram that follows depicts a main program with three functions: `one()`, `two()`, and `other()`. Within the box for each function, parameters are shown above the dashed line, local variables below it. All functions return an `int` result. All variables and parameters are type `int`. The global constant, `RATE`, and global variable, `color`, also are type `int`.



For each function call shown that follows, say whether it is legal or illegal. Fix any illegal statements.

- (a) In `main()`: `f = one( RATE );`.
- (b) In `main()`, after calling `one()`: `f = two( g, k );`.
- (c) In `one()`: `n = two( m );`.
- (d) In `one()`: `n = two( color, m );`.
- (e) In `one()`: `n = other( k );`.
- (f) In `two()`, after being called from `one()`: `r = other( k );`.

### 9.7.3 Using the Computer

1. A function.

Write a function to compute the formula

$$f(x) = (3x + 1)^{\frac{1}{2}}$$

Write a main program that will sum  $f(x)$  for the values  $x = 0$  to  $x = 1000$  in steps of 50. Print out the value of  $f(x)$  and the current sum at every step in a nice neat table.

2. Tables.

Write a program that contains two function definitions:

$$\begin{aligned} f1(n, x) &= e^{\sqrt{nx}} \times \sin(nx) \\ f2(n, x) &= e^{\sqrt{nx}} \times \cos(nx) \end{aligned}$$

where  $n$  is an integer and  $x$  is a `double`. In the main program, input a value for  $x$  and restrict it to the range  $0.1 \leq x \leq 2.5$ . Print a neat table showing the values of  $f1(n, x)$  and  $f2(n, x)$  as  $n$  goes from 0 to 30. Print column headings above the first line. Show all numbers to three decimal places.

## 3. Bubbles.

Modify your program from computer exercise 3 in Chapter 7; change the `bubble()` function so that it takes both  $\sigma$  and  $r$  as parameters. Then change the `main()` function to ask the user to enter a value for  $\sigma$  as well. Define an error function that prints a message “Input out of range.” Use it to screen out values of  $\sigma$  less than 0.001 or greater than 0.003 lb/ft and values of  $r$  less than 0.0002 or greater than 0.015 ft. Call this function from `main()`.

## 4. An arithmetic series.

Each term of an arithmetic series is a constant amount greater than the term before it. Suppose the first term in a series is  $a$  and the difference between two adjacent terms is  $d$ . Then the  $k$ th term is  $t_k = a + (k - 1)d$ . Write a function `term()` with three parameters,  $a$ ,  $d$ , and  $k$ , that will return the  $k$ th term of the series defined by  $a$  and  $d$ . Use type `long int` for all variables. Write a program that prompts the user for  $a$  and  $d$ , then prints the first 100 terms of the series, with 5 terms on each line of output, arranged neatly in columns. Quit early if integer overflow occurs.

## 5. A geometric series.

A *geometric progression* is a series of numbers in which each term is a constant times the preceding term. The constant,  $R$ , is called the *common ratio*. If the first term in the series is  $a$ , then succeeding terms will be  $aR$ ,  $aR^2$ ,  $aR^3$ , ... The  $k$ th term of the series will be  $t_k = aR^{k-1}$ . Write a function `term()` with three parameters ( $a$ ,  $R$ , and  $k$ ) that will return the  $k$ th term of the series defined by  $a$  and  $R$ . Use type `double` for all variables, since the terms grow large rapidly when  $R$  is greater than 1. Write a main program that prompts the user for  $a$  and  $R$ , then prints the terms of the series they define until either 50 terms have been printed, 5 terms per line, or floating-point overflow or underflow occurs.

## 6. Torque.

Given a solid body, the net torque  $T$  (Nm) about its axis of rotation is related to the moment of inertia  $I$  and the angular acceleration  $acc$  (rad/s<sup>2</sup>) according to the formula for the conservation of angular momentum:

$$T = I \cdot acc$$

The moment of inertia depends on the shape of the body; for a disk with radius  $r$ , it is

$$I = 0.5mr^2$$

- (a) Write a function named `moment()` to calculate and return the moment of inertia of a solid disk. It should take two parameters: the radius and mass of the disk.
- (b) Write a `work()` function that will prompt the user to enter the radius, mass, and angular acceleration of a disk. Limit the inputs to be within these ranges:

$$0.093 \leq r \leq 0.207$$

$$0.088 < m \leq 11$$

Compute and print the torque of the disk, calling `moment()` to compute  $I$  first.

- (c) Write a main program that will allow the user to compute several torques.

## 7. An AC circuit.

An AC circuit that you have designed is operating at a voltage  $V$  (rms volts) alternating at a frequency  $f$  (cyc/s). It is constructed of a capacitor  $C$  (farads), inductor  $L$  (henrys), and a resistor  $R$  (ohms) in series. Some important properties of this circuit are

Impedance:  $Z = \left[ R^2 + \left( 2\pi fL - \frac{1}{2\pi fC} \right)^2 \right]^{0.5}$  (ohms)

Current:  $I = \frac{V}{Z}$  (rms amps)

Power used: Power =  $\frac{V \times I \times R}{Z}$  (watts)

Write a function to compute an answer for each formula. Assume that  $f$  is a constant 120 cyc/s and  $C = 0.00000001$  farads. Then write a main program that will input values for  $V$ ,  $L$ , and  $R$  and output the impedance, current, and power used. Validate the inputs and ensure that they are within the following ranges:

$$60 \leq V \leq 200$$

$$0.1 \leq L \leq 10$$

$$100 \leq R \leq 1000$$

#### 8. Ice cream cones.

Suppose you are a professional party planner. Given the number of guests expected, you must plan a menu and deliver enough food to serve the crowd. In this problem, you will write a program to calculate how many packages of ice cream must be bought to fill one ice cream cone for each guest. The guest will select the size of the cone (diameter, height). Your suppliers sell ice cream in containers of various sizes and shapes.

Write a function named `cone()` that will prompt for and read the diameter,  $d$ , and the height,  $h$ , of the cone to be used, then calculate and return the volume of ice cream needed to fill the cone. Assume that the cone part will be filled entirely and there will be a hemisphere of ice cream on top. The formulas are:

$$\text{Volume of cone} = \frac{\pi \times d^2 \times h}{12}$$

$$\text{Volume of hemisphere} = \frac{\pi \times d^3}{12}$$

Ice cream comes in cartons that are either the shape of a barrel or a box. Write a function named `carton()` that will prompt for an alphabetic code,  $R$  for “barrel” or  $X$  for “box”. Use a switch statement to execute the appropriate input and computation instructions, then return the volume of the selected carton. For a barrel, input the diameter and height; for a box, input the length, width, and height. Use one of these formulas:

$$\text{Volume of barrel} = \frac{\pi \times \text{diameter}^2 \times \text{height}}{4}$$

$$\text{Volume of box} = \text{length} \times \text{width} \times \text{height}$$

In your main program, prompt for and input the number of guests, then call your `cone()` function and your `carton()` function. Finally, compute and display the number of cartons of ice cream you must buy to fill all those cones. Use the `ceil()` function to round up to a whole number of cartons.

#### 9. Wedding cake.

Another common party food is wedding cake. Given the number of guests expected at the wedding, write a program to calculate how many layers your cake must have to serve everyone, and how much that cake will cost.

The cake will be square and have two or more tiers. The top tier will be 6” wide and will not be eaten at the wedding reception. Each other tier will be 2” thick and 4” wider than the tier above it. Guests will be served from layers 2, 3, 4...; each serving will be 2” square.

Write the following one-line functions:

- `width()`: calculate the width of a tier given the layer number (layer 1=6”, layer 2 = (6+4)”, layer 3 = (6+4+4)”, etc.).
- `servings()`: calculate the number of servings a tier will provide, given the layer number. Call `width()`.
- `price()`: the top tier and decorations cost \$40.00; the other tiers cost \$1.00 per serving. (This will be more than \$1.00 per guest because part of the bottom tier will be left over.)

Write a function named `layers()` that will calculate how many tiers are needed. Hint: start with 1 tier, which serves 0 people. Use a loop to call `servings()` and add tiers until you have accumulated enough portions to serve the party. In general, you will end up with more than enough servings, since part of the last tier will be left over.

In your main program, prompt for and read the number of guests, then call the `layers()` and `price()` functions to calculate the size and cost of the cake. Print out these answers.

10. Scheduled time of arrival.

Airline travelers often want to know what time a flight will arrive at its destination in the local time of the destination. This can be calculated given the following data:

- The scheduled takeoff time, in hours and minutes on a 24-hour clock. (Valid hours are 0...23, valid minutes are 0...59)
- The scheduled duration of the flight, in hours and minutes.
- The number of time zone boundaries the flight will cross. This number should be negative if traveling from East to West, positive if going West to East.
- Whether the international date line will be crossed. This number should be +1 if it is crossed traveling from East to West, -1 if crossed while going West to East, and 0 if it is not crossed. Legal values are in the range -23...23.

Using top-down design, write a program with functions that will make this calculation for a series of flights. For each flight, your program must input these data values from the user and print the scheduled time at which the flight should arrive at its destination. This time is calculated as follows:

- Starting with the takeoff time, add or subtract an hour for each time-zone change.
- Then add the duration of the flight to this time.
- Finally, adjust the time by adding or subtracting a day if the flight crossed the international date line.
- Use integer division and the modulus operator to convert minutes to hours + minutes, and hours to days + hours.

Print the local time of arrival using a 24-hour clock. Also print `-1 day` if the flight will land the day before it took off or `+1 day` if it will land the day after it took off (both are possible).

Suggestion: Write a function that will input, validate, and return one integer. It should have two integer parameters: the minimum and maximum acceptable values for the input.



# Chapter 10

## An Introduction to Arrays

The data types we have studied so far have been simple, predefined types and variables of these types are used to represent single data items. The real power of a computer language, however, comes from its ability to define complex, multipart, structured data types that model the complex properties of real-world objects. For example, to model the periodic table of the elements, we would need a collection of 110 objects that represent elements; each object would have several parts (name, symbol, atomic number, atomic weight, etc.). We call such types *compound types* or **aggregate types**. An array is an aggregate whose parts are all the same type. In this chapter, we study how to define, access, and manipulate the elements of an array. The last half of the chapter presents simple, important array algorithms.

### 10.1 Arrays

In many applications, each data item is processed once, just after it is read, and never needs to be used again. In such programs, an array can be used to store the data, but it is not necessary. In contrast, some programs must read all the data, perform a calculation, then go back and process the data again. In these programs, we must store all the data between reading it and reprocessing it.

Consider the problem of assigning grades to a class based on the average score of an exam. If we were computing only the exam average, this could be done without storing the data in an array; all that is needed is a summation loop. However, to assign a grade, we must have both the exam score and the average. The scores must be processed once to compute the average, then we must go back and reprocess the scores to assign the grades. In between we need to store the values.

We could do this using individual variables. For instance, in the example of computing the average of three numbers in Figure 2.7, we used three separate variables to hold the numbers. While this works well for only three numbers, it does not work on larger data sets. Imagine how tedious it would be to write a `scanf()` statement and a formula with many variable names. We can solve this problem by using an array. For example, a program to compute the average temperature over a 24-hour period might store 24 temperature readings in an array named `temperature`. An array used to determine the average high temperature for one year would have 365 (or 366) entries, each containing a daily high temperature.

An **array** is a consecutive series of variables that share one variable name. We will call these variables the **array slots**; the data values stored there are called the **array elements**. The number of slots in a *one-dimensional array* is called the **array length**. Arrays with two or more dimensions also can be defined; these will be studied in Chapter 18. The variables, or slots, that form an array have a uniform type called the **base type**.

An array object, as a whole, is given a name. We can refer to the entire array by this name or to an individual slot by appending a number in square brackets to the name. This number is called the *subscript*. A **subscript** is an integer expression enclosed in square brackets that, when written after an array name, designates a particular slot in the array.

In C, all arrays start with slot 0 (rather than 1) because it is easier and more efficient for the system to implement. This means that the first element in an array named `ary` would be called `ary[0]`; the next one would be `ary[1]`. If this array had six elements, the last one would be `ary[5]`.

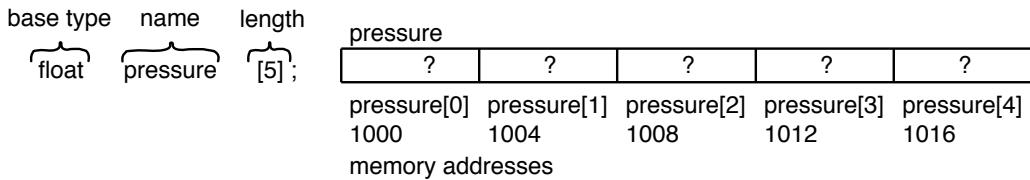


Figure 10.1. Declaring an array of five floats.

### 10.1.1 Array Declarations and Initializers

**Declarations.** An array variable is declared and initialized very much like a simple variable. The **array declaration** starts with the base type, which can be any type—simple or compound. Thus, we can have an array of **ints**, an array of **chars** (also known as a *string*), or even an array of arrays. Following the base type in the declaration is the array name and a pair of square brackets enclosing the length, which must be an integer constant or an integer **constant expression** (an expression with only constant operands). The length determines the number of slots in the array. Figure 10.1 shows the declaration for an array named **pressure** containing five **floats**. This creates a series of five **float** variables that we can refer to as **pressure[0]**, **pressure[1]**, **pressure[2]**, **pressure[3]**, and **pressure[4]**. These five **floats** will be stored in a contiguous set of memory locations with **pressure[0]** at the location with the lowest memory address, 1000, in this case. In later chapters, the address of each slot may also be of interest; if so, we write the address above or below the slot, as shown in Figure 10.1.

To diagram an array, we draw a row of connected boxes that are the right size for the base type. We write the name of the array above and the subscripts below this row. If there is an initializer, as in Figure 10.2, we copy the initial values into the boxes. Otherwise, as in Figure 10.1, we leave them blank or write a question mark.

**Initial values.** An array can be declared with no initial values, like the array named **pressure** in Figure 10.1. The contents of an uninitialized array are as unpredictable as ordinary variables.<sup>1</sup>

Alternatively, it may have an **array initializer**, which is a list of values enclosed in curly brackets. The values will be stored in the array slots when the array is created, as illustrated in Figure 10.2. The values in the initializer list must be constants or constant expressions. The types of values in the initializer must be appropriate for the base type of the array.<sup>2</sup>

If an initializer *is* given, the array length may be omitted from the square brackets. The C translator will count the number of initial values given and use that number as the length; exactly enough space will be allocated to store the given values. Note the absence of the length value in the declaration for **temperature** in Figure 10.3.

<sup>1</sup>Global arrays and static local arrays are initialized to 0 values.

<sup>2</sup>The initializer type must match or be coercible to the array base type.

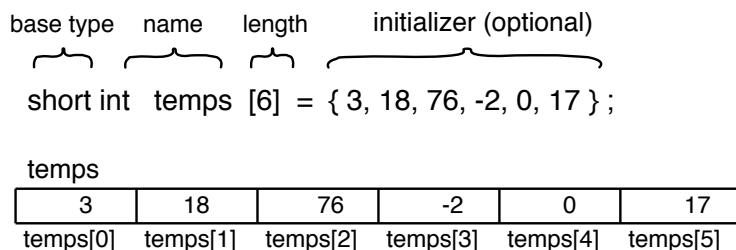


Figure 10.2. An initialized array of short ints.

base type	name	length	initializer (optional)	
short int	age	[6]		age
				[0] [1] [2] [3] [4] [5]
			temperature	3 18 76 -2 0 17
				[0] [1] [2] [3] [4] [5]
			inventor	Y 1 0 0 0 0 0
				[0] [1] [2] [3] [4] [5]
			instock	0 0 0 0 0 0
				[0] [1] [2] [3] [4] [5]

Figure 10.3. Length and initializer options.

What if both a length and an initializer are given and the sizes do not match? This is an error if the initializer contains too many values; the compiler will detect this error and comment on it. However, if an initializer list is too short, it is not an error. In this case, the values provided are used for the first few array slots and the value 0 is used to initialize all remaining slots. The declaration for `inventory` on the third line in Figure 10.3 illustrates an initializer that is shorter than the array. It also is possible to initialize an entire array to zeros by supplying an empty set of brackets, as in the initializer for `instock` in Figure 10.3.

**Arrays of characters.** There are two ways to initialize an array of characters: with a comma-separated series of char literals, or with a string literal, as shown in Figure 10.4. It is certainly much easier to use a one string literal than to write a lot of pairs of single quote marks. Note, however, that the string initializer puts one extra character into the array that is not visible in the code: a null character that marks the end of the string. This will be explained in more detail in Chapter 12.

### 10.1.2 The Size of an Array

Two different aspects of an array's size are important: its length and the total number of bytes of memory required to store it. When we use `sizeof` with an array variable, we get the number of bytes, which is the product of the array's length and the size of one element of its base type.

While knowing the number of slots is necessary to write an array declaration, a programmer will not always know the **size of the array**, because that depends on the size of the base type, which may vary from one machine to another. The program in Figure 10.5 shows the sizes of the arrays declared in Figures 10.1 and 10.2, and two arrays of chars. An output from this program is

```
pressure: sizeof(float) is 4 * length 5 = sizeof array 20
temps: sizeof(short) is 2 * length 6 = sizeof array 12
```

vowels					
char	vowels[]	=	{'a','e','i','o','u'};	'a'	'e'
operators					
char	operators[]	=	{"+-*/%"};	'+'	'-'
				'*'	'/'
				'%'	'\0'
				[0]	[1]
				[2]	[3]
				[4]	[5]

Figure 10.4. Initializing character arrays.

```
#include <stdio.h>

#define DIMP 5      // Lengths of the arrays.
#define DIMT 6
#define DIMV 3

int main( void )
{
    float pressure[DIMP] = { .174, 23.72, 1.111, 721.2, 36.3 };
    short int temps[DIMT] = { 3, 18, 76, -2 };
    double vec2[DIMV] = { 2.0, 0.0, 1.0 };
    char vowels[] = 'a','e','i','o','u';
    char operators[] = "+-*%/";

    printf( " pressure: sizeof(float) is %i * length %i ="
           " sizeof array %i \n", sizeof(float), DIMP, sizeof(pressure) );
    printf( " temps:     sizeof(short) is %i * length %i ="
           " sizeof array %i \n", sizeof(short), DIMT, sizeof(temps) );
    printf( " vec2:      sizeof(double) is %i * length %i ="
           " sizeof array %i \n", sizeof(double), DIMV, sizeof(vec2) );
    printf( " vowels:    sizeof(char)   is %li sizeof vowels is %li\n",
           sizeof(char), sizeof(vowels) );
    printf( " operators: sizeof(char)   is %li sizeof operators is %li \n",
           sizeof(char), sizeof(operators) );
    return 0;
}
```

---

**Figure 10.5.** The size of an array.

```
vec2:     sizeof(double) is 8 * length 3 = sizeof array 24
vowels:   sizeof(char)   is 1   sizeof vowels is 5
operators: sizeof(char)   is 1   sizeof operators is 6
```

Often, the first portion of an array will hold data, while the last portion is not in use. This happens when the array is intended to hold a variable amount of information and its length is set to the maximum length that might ever be needed. Even in this case, the size returned by `sizeof` is the total amount of memory allocated including both the portion in use and the unused portion. This is illustrated by the array `temps` in Figure 10.5.

When an array is passed as an argument to a function, (see Section 10.4) the size information does not travel along with it. No operation in C will give us the actual length of an array argument inside a function. If you apply `sizeof` to an array parameter, the result always will be the number of bytes needed to store the starting address of the array. Unfortunately, an array-processing function frequently needs the information to work properly. For this reason, the programmer, who knows the array's length when it is declared, must make the information available for use by every part of the program that operates on the array. One way to do this is to declare the array length using a `#define` at the top of the program, as in our first several examples.

### 10.1.3 Accessing Arrays

The elements of an array can be accessed in two ways: by using pointers or by using subscripts. Both ways are important in C and need to be mastered. Pointers often are used when the slots of an array will be used in sequential numeric order, because this technique can lead to greater efficiency. While subscripts can be used for this purpose, too, they are better at accessing the elements in a random order. Since pointer processing techniques are harder to master than those based on subscripts, using pointers will be deferred until Chapter 16, while subscripting is explained in this chapter.

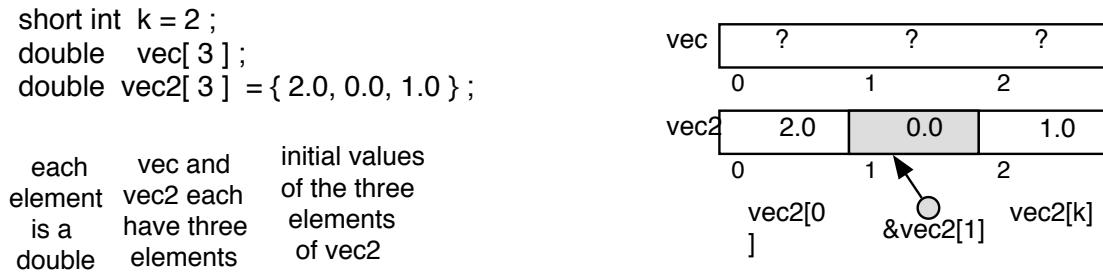


Figure 10.6. Simple subscripts.

**Subscripts.** Figure 10.6 shows how constant subscripts are interpreted. It shows two arrays named `vec` and `vec2`, which represent vectors in a three-dimensional space. The first slot of `vec2` is `vec2[0]` and represents the  $x$ -component of the vector, containing the value 2.0. The  $y$ -component has the value 0.0 and is stored in `vec2[1]`, the shaded area in the diagram. In an object diagram, we use an arrow to represent an address. The arrow in Figure 10.6 represents `&vec2[1]`, the address of the shaded area. When both an ampersand and a subscript are used, the subscripting operation is done first and the “address of” operation is applied to the single variable selected by the subscript.<sup>3</sup> These addresses are important when using arrays as function arguments, as demonstrated in Section 10.4.

We also can use an expression involving a variable to compute a subscript (Figure 10.7). A subscript expression can be as simple as a constant or as complex as a function call, as long as the final value is a nonnegative integer less than the length of the array. The most frequently used subscript expression is a simple variable name, which also is illustrated in Figure 10.6. The phrase `vec2[k]` means that the current value of `k` should be used as a subscript for the array. Since `k` contains a 2 here, `vec2[k]` means `vec2[2]`, which contains the value 1.0. The flexibility and versatility of these subscript rules enables a variety of powerful array-processing techniques.

**Notes on Figure 10.7. Computed subscripts.** Figure 10.6 demonstrates how we can use computed subscripts. It depicts an array containing the ages of  $N$  members of a family: father, mother, and children, in order of age. The array length is #defined. Often this is done so that it can be used for calculations throughout the code yet modified easily if the program’s needs change.

The variable `pos_mother` is used here to store the position in the array of the mother. In the last line on the left, the expression `pos_mother+1` is used to find the age of the oldest child. Also, we often need to use the subscript of the last element in an array. To compute this, we subtract 1 from the array length. Therefore, `age[N-1]` is the age of the last (youngest) child in the family.

<sup>3</sup>Appendix B contains a precedence table that includes both subscript (`[]`) and address of (`&`) operators. Note that the precedence of subscript is higher, and therefore, the subscript will be applied to the array name before the address operator.



Figure 10.7. Computed subscripts.

---

This brief demonstration program shows how to do input, output, and computation with the elements of an array.

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

int main( void )
{
    double vec[3];      // A vector in 3-space.
    double magnitude;

    puts( "\n Subscript Demo: The Magnitude of a Vector" );

    printf( " Please enter the 3 components of a vector: " );
    scanf( "%lg%lg%lg", &vec[0], &vec[1], &vec[2] );

    magnitude = sqrt( vec[0] * vec[0] + vec[1] * vec[1] + vec[2] * vec[2] );

    printf( "     The magnitude of vector ( %g, %g, %g ) is %g \n",
            vec[0], vec[1], vec[2], magnitude );
    return 0;
}
```

---

**Figure 10.8. Subscript demo, the magnitude of a vector in 3-space.**

The example in Figure 10.8 declares an array variable named `vec` and shows simple input, computation, and output statements that operate on the array elements.

#### Notes on Figure 10.8. Subscript demo, the magnitude of a vector.

**First box: input into an array.** When we call `scanf()` to read the value of a variable, we write `&` before the variable's name to refer to its address. Similarly, we can use `&` to refer to the address of a single slot of an array. To read just one value into the first component, we would write `scanf( "%lg", &vec[0] );`.

**Second box: computation on array elements.**

- We can use a subscripted array name like a simple variable name in any expression.
- Here, to compute the magnitude of a vector, we add the squares of the three components, take the square root of the sum, and store the result in `magnitude`. The easiest and most efficient way to square a number is to multiply it by itself.

**Third box: output from an array.**

- To print an array element, give the array name and the subscript of the element.
- You cannot print the entire contents of an array with just one format field specifier. Here, we use three separate `%g` specifiers to print the three array elements. A loop would be used to print all the elements of a large array,
- The output from two runs of this program is

```
Subscript Demo: The Magnitude of a Vector
Please enter the 3 components of a vector: 1 0 1
The magnitude of vector ( 1, 0, 1 ) is 1.41421
-----
```

```
Subscript Demo: The Magnitude of a Vector
Please enter the 3 components of a vector: 1 2 0
The magnitude of vector ( 1, 2, 0 ) is 2.23607
```

```

#include <stdio.h>
#define N 3

int main( void )
{
    float dimension[N];      // Dimensions of a box.
    float volume;            // The volume of the box.

    printf( "\n Array Input Demo: the Volume of a Box \n"
            " Please enter dimensions of box in cm when prompted.\n" );

    for (int k = 0; k < N; ++k) { // End loop when k reaches length of array.
        printf( " > " );
        scanf( "%g", &dimension[k] );
    }

    volume = dimension[0] * dimension[1] * dimension[2] / 1e6;
    printf( " Volume of the box ( %g * %g * %g ) is %g cubic m.\n\n",
            dimension[0], dimension[1], dimension[2], volume );
    return 0;
}

```

Figure 10.9. Filling an array with data.

#### 10.1.4 Subscript Out-of-Range Errors

One important reminder and caution: It is *up to the programmer* to ensure that all subscripts used are legal. C does not help you confine your processing to the array slots that you defined. C uses the subscript value to compute a memory address called the **effective address** according to this formula:

$$\text{effective\_address} = \text{address of beginning of the array} + \\ \text{subscript} \times \text{sizeof}(\text{base type of array})$$

If you use a subscript that is negative or too large, C will compute the theoretical “address” of the nonexistent slot and use that address even though it will not be between the beginning and the end of the array. C does absolutely no subscript range checking. The compiler will give no error comment and there will be no error comment at run-time either. Your program will run and either access a memory location that belongs to some other variable or attempt to access a location that does not exist.

## 10.2 Using Arrays

A common array processing pattern involves a counting loop, where the loop variable starts at 0 and stops before  $N$ , the length of the array. The loop variable is used as a subscript to access each array element, in turn. This kind of loop is seen again and again in programs that use arrays to process large amounts of data. Often, one loop is used to read data into the array, another to process the data items, and a third loop to print them all.

### 10.2.1 Array Input

The best way to read data into an array is with a **for** loop, where the loop counter starts at 0, increments through the subscripts of the array, and ends at  $N$ , the declared length of the array. The loop does not try to process slot  $N$ , which is past the end of an  $N$ -element array. This simple idiom is illustrated in Figure 10.9.

**Notes on Figure 10.9. Filling an array with data.**

---

This program is a modification of the vector magnitude program in Figure 10.8. It demonstrates how an incorrect loop can destroy the value of a variable and result in unpredictable behavior.

```
#include <stdio.h>
int main( void )
{
    int k;           // Loop counter.
    float v[3];      // A vector in 3-space.
    float sum;        // The sum of the squares of the components.
    float magnitude;
    puts( "\n Subscript Demo: Walking on Memory" );
    printf( " Please enter one float vector component at each prompt.\n" );

    for (sum = 0.0, k = 0; k <= 3; ++k) {          // Loop goes too far.
        printf( "\tv[%i]: ", k );
        scanf( "%g", &v[k] );
        sum += v[k] * v[k];
    }
    magnitude = sqrt( sum );
    printf( "      The magnitude of vector ( %g, %g, %g ) is %g \n",
           v[0], v[1], v[2], magnitude );
    return 0;
}
```

---

**Figure 10.10. Walking on memory.**

**Outer box: the for loop.** Since the loop counter takes on the values from 0 to N and the loop ends when  $k == N$ , all N array slots (with subscripts  $0\dots N-1$ ) will be filled with data.

**Inner box: reading data into one slot.** Within the loop, a `scanf()` statement reads one data value on each iteration and stores it in the next array slot. Note that both an ampersand and a subscript are used in the `scanf()` statement to get the address of the current slot.

*Sample output.*

```
Array Input Demo: the Volume of a Box
Please enter dimensions of box in cm when prompted.
> 100
> 100
> 100
Volume of the box ( 100 * 100 * 100 ) is 1 cubic m.
```

### 10.2.2 Walking on Memory

A loop that runs amok can cause diverse kinds of trouble. One common outcome is that variables that occupy nearby memory locations are overwritten with information that was supposed to go into one of the array's slots. Afterward, any computation or output that uses these variables will be erroneous.

If you write a loop to print the values in an array and it loops too many times, you will start printing the values stored adjacent to the array in memory. When storing data, you will start erasing other information when the loop exceeds the array bounds. This is demonstrated by the program in Figure 10.10 (a modification of the program in Figure 10.8), which reads input into an array named `v`. Figure 10.11 is a diagram of memory for this program. It shows the variables and the memory addresses that a typical compiler might assign. The array is colored gray.

The upper diagram in Figure 10.11 shows the values of the variables just after the third trip around the loop. All the array slots have been filled, and `k` has been incremented to 3 and is ready for the loop exit test.

These diagrams illustrate the contents of memory while processing the input sequence described in the text.

After filling the array on the third trip around the loop:



After exceeding array bounds on the fourth trip around the loop:



Running amok, incrementing *k* before the fifth trip around the loop:



**Figure 10.11. Before and after walking on memory.**

However, since the test was written incorrectly (with a  $\leq$  operator instead of a  $<$  operator), the loop does not end.

The array has three slots, but the loop was written to execute four times, with subscripts 0 through 3. The last time, the effective address calculated for  $v[k]$  actually is the address of  $k$ , and the input value, wrongly, is stored on top of the loop counter. The output from this run had more lines than expected:

```
Subscript Demo: Walking on Memory
Please enter one float vector component at each prompt.
v[0]: 0.0
v[1]: 1.0
v[2]: 2.0
v[3]: 0.0
v[1]: 5.0
v[2]: -1.0
v[3]: 4.5
Segmentation fault
```

The middle diagram shows what happens during the next loop iteration, as the subscript goes beyond the end of the array. The input value of 0.0 is stored in the variable that follows the array, which is the memory location used for the loop counter. This destroys the value of the loop counter and leaves the input value in its place. In this example, the input is 0, which then gets incremented to 1 by the `for` loop (bottom diagram). The loop still does not terminate, because now  $k$  is 1. It continues taking input until some input value is stored in  $k$  that satisfies the loop exit test or until an abnormal termination happens, as occurs here.

In this example, storing input on top of the loop counter causes unpredictable behavior that depends on the data entered by the user. Usually, as in the output shown, the program crashes; other times it continues and terminates normally but produces erroneous results. Be sure to check the limits of array processing loops to minimize these problems.

**Random locations and memory faults.** Sometimes, a faulty subscript causes the program to try to use a memory location that is not legal for that program to access; the result is an immediate hardware error (a memory fault, bus error, or segmentation fault) that terminates the program. Usually, the system displays a message to this effect, as was seen in the last program.

---

**Problem scope:** Given the ID number and two exam scores (midterm and final) for each student in a class, compute the weighted average of the two scores. Also, compute the overall class average and the difference between that and each student's average.

**Input:** ID numbers will be long integers and exam scores will be integers.

**Limitations:** The class cannot have more than 16 students.

**Formula:** Weighted average =  $0.45 \times \text{midterm score} + 0.55 \times \text{final score}$

**Output and computational requirements:** All inputs should be echoed. In addition, for each student, print the exam average and the difference between that average and the overall average for the class, both to one decimal place. The overall exam average of the class should be printed using two decimal places.

---

**Figure 10.12. Problem specifications: Exam averages.**

If a program continues to run after a subscript error, it generally produces erroneous output. The cause of the errors may be difficult to detect because the value of some variable can be changed by a part of the program that does not refer to that variable at all, and output based on the mistake may not occur until long after the destructive deed. Prevention is the best strategy for developing working code. It is up to the programmer to use subscripts carefully and ensure that every subscript is legal for its array. With this in mind, remember that

- Arrays start with subscript 0, so the largest legal subscript is one less than the number of elements in the array.
- An input value must be checked before it can be used as a subscript. Negative values and values equal to or larger than the array length must be eliminated.
- A counting loop that processes an array should terminate when the loop counter reaches the number of items in the array.

### 10.3 Parallel Arrays

The next program (specified in Figure 10.12 and given in Figure 10.13) uses a set of **parallel arrays**, all of the same length, to implement a table of data. Each array in the set represents one column of the table and each array subscript represents one row of data. In this example, the first column is a list of student ID numbers. Parallel to it are three other arrays containing data about the students. The data at subscript  $k$  in each of these arrays corresponds to the student with subscript  $k$  in the ID array. This is illustrated by the declarations and diagram in Figure 10.14.

When a table is implemented as a set of parallel arrays, the same variable is used to subscript all of them. We can apply this principle here. A loop is used to select the array slots. For each slot, first, input is read into three of the arrays at the selected position, then an average is calculated and stored in the fourth array. After the input loop, we scan this last array to compute several more values.

#### Notes on Figure 10.13. Using parallel arrays.

##### *First box: limiting the subscripts.*

- Serious errors result from using a subscript beyond the end of the array. To avoid this, we check that the class size is within the limits we are prepared to handle. If it is too large, we abort. We also abort if the size is negative or 0, because these values are meaningless.
- If a class really had more than 16 students, this program would need to be edited to make **MAX** larger and then recompiled, so we print an error comment and end execution gracefully.

##### *Second box: the input phase.*

- Our input loop counts from 0 up to the class size the user has entered. Since this count has been validated, we can be sure that all array subscripts are legal.
- On each iteration we enter all the data for one student. Three numbers are read and stored in the corresponding slots of the first three arrays.

```

#include <stdio.h>
#define MAX 16
int main( void )
{
    int n;                      // Number of students in class.
    long id[MAX];              // Students' ID numbers.
    short midterm[MAX], final[MAX]; // Exam scores.
    float average[MAX];         // Average of exam scores.
    float avg_average;          // Average of averages.
    float diff;                 // Student's average minus class average.

    printf( " Exam average = .45*midterm + .55*final.\n"
            " How many students are in the class? " );
    scanf( "%i", &n );
    if (n > MAX || n < 1) {
        printf( "Size must be between 1 and %i.", MAX );
        exit(1);
    }

    printf( " At each prompt, enter an ID# and two exam scores.\n" );
    for (int k = 0; k < n; ++k) {
        printf( "\t> " );
        scanf( "%li%hi%hi", &id[k], &midterm[k], &final[k] );
        average[k] = .45 * midterm[k] + .55 * final[k];
    }

    for (avg_average = 0, k = 0; k < n; ++k) avg_average += average[k];
    avg_average /= n;
    printf( "\nAverage of the averages = %.2f\n", avg_average );

    puts( "\nID num    mid    fin    average    +/- " );
    puts( "-----" );
    for (int k = 0; k < n; ++k) {
        diff = average[k] - avg_average;
        printf( "%li %5hi %5hi %8.1f %6.1f \n",
                id[k], midterm[k], final[k], average[k], diff );
    }
    puts( "-----" );

    return 0;
}

```

Figure 10.13. Using parallel arrays.

This is a diagram of the memory for the program in Figure 10.13. A set of parallel arrays is used to represent the exam scores and exam average for a class. A common subscript, *k*, is used to subscript all four arrays. The maximum number of students this table can hold is 16, but this class has only 13 students, so the last three array slots are empty.

			id	midterm	final	average
#define MAX 16	MAX: 16	0	825176	80	85	82.8
int k;	k	1	825301	72	68	69.8
int n;	3	2	824769	97	90	93.2
long id[MAX];	n	3	826162	57	66	62.0
short midterm[MAX];	13	4	824388	88	92	90.2
short final[MAX];		5	825564	42	61	52.5
float average[MAX];		6	825923	75	62	67.8
		7	823976	82	81	81.4
		8	824662	91	94	92.7
		9	824478	68	80	74.6
		10	826056	82	71	75.9
		11	826178	95	97	96.1
		12	825743	51	57	54.3
		13				
		14				
		15				

Figure 10.14. Parallel arrays can represent a table.

- Sometimes the input loop does only input and other loops are used to process the data. In this example, the loop both reads the input and calculates the weighted average, which is part of a student's record and based directly on the input. This average is stored in the fourth array (the fourth column of the table). Merging these actions leads to a slightly more efficient program. However, if the additional calculations are lengthy, efficiency can be sacrificed for the added clarity of splitting the tasks into separate loops.

- The prompts and input process look like this:

```
Exam average = .45*midterm + .55*final.
How many students are in the class? 13
At each prompt, enter an ID# and two exam scores.
> 825176 80 85
> 825301 72 68
> 824769 97 90
> 826162 57 66
> 824388 88 92
> 825564 42 61
> 825923 75 62
> 823976 82 81
> 824662 91 94
> 824478 68 80
> 826056 82 71
> 826178 95 97
> 825743 51 57
```

- We will echo the input data later, along with calculated values.

#### *Third box: the average calculation.*

- We sum the weighted averages as the first step of computing the overall class average. This task also could have been done as part of the input loop but was written as a separate loop because it has no direct

connection to the input process or a single student's record.

- We use a one-line `for` loop, because summing the values in an array is a simple job that corresponds to a single conceptual action.
- Note how convenient the `+=` operator is for summing the elements of an array. The `/=` operator provides a concise way to say "now divide the sum by the number of students."
- The output from this box is

Average of the averages = 76.40

**Fourth and fifth boxes: the output phase.**

- In the outer box, we print table headings before the output loop and print a line to terminate the table after the loop.
- In the inner box, we print each student's record by including one value from each of the parallel arrays and a final value computed from the average array. Note that we use `%f` in the `printf()` format to make nicely aligned columns.
- The input data is printed side by side with the final output to make it easier to check whether the computations are correct.
- The final output of the program is:

ID num	mid	fin	average	+/-
<hr/>				
825176	80	85	82.8	6.3
825301	72	68	69.8	-6.6
824769	97	90	93.2	16.8
826162	57	66	62.0	-14.5
824388	88	92	90.2	13.8
825564	42	61	52.5	-24.0
825923	75	62	67.8	-8.6
823976	82	81	81.4	5.0
824662	91	94	92.7	16.2
824478	68	80	74.6	-1.8
826056	82	71	75.9	-0.5
826178	95	97	96.1	19.7
825743	51	57	54.3	-22.1
<hr/>				

## 10.4 Array Arguments and Parameters

No data type is very useful in a programming language unless it can be used in a function call to pass information into and out of a function. Therefore, we need to know how to write a function with an array parameter and how to call such a function with an array argument. C does not permit a function to *return* an array value.<sup>4</sup>

Array arguments in C are handled differently from other types of arguments. When an `int`, a `double`, or a single element from an array is passed to a function, its value is copied into the parameter variable that has been created for the function. Technically, we say that arguments of simple types are passed *by value*. However, when an array is passed to a function, it is passed *by reference*, that is, only the *address* of the first slot of the array, not its entire list of values, is copied into the function's memory area. This is similar to the way that `scanf()` works. The address of a variable is passed to `scanf()`, which fills it with information from the keyboard, and this information remains in the variable even after `scanf()` finishes.

Passing array arguments by reference permits a large amount of data to be made available to a function efficiently (since the actual data are not copied) and also allows the function to store information into the array. Therefore, a program can pass an empty array into a function, which then will fill it with information. When the function returns, that information still is in the original array's memory and can be used by the caller.

To call a function with an **array argument**, the caller simply writes the name of the array and does *not* write a subscript or the square subscript brackets. Also, no `&` operator is used in front of the array name,

---

<sup>4</sup>However, it is possible to return a pointer to an array. This topic is deferred until a later chapter.

because the array name automatically is translated into its starting address. To declare the corresponding **array parameter**, however, we use empty square brackets (with no length value). The length may be written between the brackets but it will be ignored by the compiler. This is done in C so the function can be used with arrays of many different lengths.

For example, if the actual argument were an array of **doubles**, a formal parameter named **ara** would be declared as **double ara[]**. Within the function, the parameter name is used with subscripts to address the corresponding argument values.

The next program illustrates the basic array operations described so far: input, output, access, calculation, and the use of an array parameter. In it, the term **FName** means function name and **AName** means array name. We introduce and demonstrate the use of three new forms of function prototypes that manipulate arrays:

- **void FName( double AName[], int n );**

A prototype of this form is used when the purpose of the function is to read data into the array or print the array data. The **get\_data()** function in the next example has this form; its prototype is

```
void get_data( double x[], int n );
```

This function takes two parameters, an array of **doubles** called **x** and an integer that gives the length of the array. Since the declaration of the parameter **x** contains no length, we need a limiting value. This can be the globally defined constant that was used to declare the array object. Often, though, we do not use the entire array, and a parameter is used to communicate the amount of the array currently in use. The **get\_data()** function fills the array with input values that remain in it after the function returns. This is one way of returning a large number of values from a function to the calling program. Since there is no other return value, the return type is declared as **void**.

- **double FName( double AName[], int n );**

A prototype of this form is used when an array contains data and we wish to access those data to calculate some result, which then is returned. The function named **average()** in the next example has this form; its prototype is

```
double average( double x[], int n );
```

It again takes two parameters, the array of **doubles** and the current length of the array. The function calculates the average (mean) of those values and returns it to the caller via the **return** statement. Therefore, the function return type is declared as **double**.

- **double FName( double AName[], int n, double Arg );**

We use a prototype of this form when we need both an array of data and another data value to perform a calculation. The function named **divisible()** in the next example has this form; its prototype is

```
int divisible( int primes[], int n, int candidate );
```

It takes three parameters, an array of prime numbers, its length **n**, and the **candidate** number we wish to test for primality. The numbers in the array are used to test the candidate; the answer will be *true* (1) or *false* (0).

## 10.5 An Array Application: Prime Numbers

The next application is a prime number generator that illustrates the use of arrays and array parameters. The task specifications are given in Figure 10.15, the main program in Figure 10.16, and a function in Figure 10.17.

### Notes on Figure 10.16. Calculating prime numbers.

#### *First box: prototype.*

- This prototype follows the third pattern discussed above: the parameters are an array, its length, and another value that must be used with the array.
- The **divisible()** function compares the **candidate** number to the numbers in the array. If the **candidate** is divisible by anything in the array, *true* (1) is returned. Otherwise it is non divisible (prime), so *false* (0) is returned.

**Problem scope:** Print a list of all prime numbers, starting with 2, and continuing until MANY primes have been printed. A prime number is an integer that has no proper divisors except itself and 1.

**Constant:** MANY, the number of primes to be found and printed.

**Restrictions:** MANY must be small enough that an array of MANY integers can fit into memory and the last prime calculated is less than the maximum integer that can be represented.

**Input:** None.

**Output required:** A neat list of primes, one per line.

Figure 10.15. Problem specifications: A table of prime numbers.

This main program calls the functions in Figure 10.17. It calculates and prints a table of the first MANY prime numbers. Strategy: identify prime numbers in ascending order and print them. Save each prime in a table and use them all to test the next number in sequence.

```
#include <stdio.h>
#define MANY 3000

void print_table( int primes[], int num_primes );
int divisible( int primes[], int n, int candidate ); // Is it nonprime?

int main( void )
{
    int k; // Integer being tested.
    int primes[MANY]={2}; // To begin, put the only even prime in table.
    int n = 1; // Number of primes currently in table.

    printf( "\nA Table of the First %i Prime Numbers\n", MANY );
    for (k = 3; n <= MANY; k += 2) { // Test the next odd integer.
        // Quit when table is full.
        if (!divisible( k, primes, n )) { // If it is a prime...
            primes[n] = k; // ... put it in the table...
            ++n; // ... and count it.
        }
    }

    print_table( primes, MANY ); // Print table of primes.
    return 0;
}
```

Figure 10.16. Calculating prime numbers.

***Second box: the table of primes.***

- The table will be filled in with prime numbers. The list will be generated in order by testing every possible odd number, starting with 3. As primes are discovered, we store them in the table. To test each integer, we use the previously computed portion of the table.
- We choose an arbitrary constant for the length of this table. Computing more primes requires more time and storage space. This method is limited by the space available and the largest integer that can be represented in the ordinary way. The latter limit usually occurs first.
- The first prime, and the only even prime, is 2. We initialize the first slot in the table to 2 so that the computation loop can be limited to testing odd numbers. The rest of the prime table will be initialized to 0.
- We already have stored one prime in the table, so we initialize `n` to 1. It will be incremented each time a new prime is found.

***Third box: filling the table.***

- We use a `for` loop that starts at 3 and counts by twos to test all the odd numbers.
- This is a very unusual loop. While we initialize `k` and increment it each time around the loop, we use `n`, the number of primes, to end the loop. We want to continue searching for primes until the table is filled; that is, `n==MANY`. Since we do not know how big `k` will be at that time, we do not use `k` to terminate the loop.

***Inner box: calling the function to test for primality.***

- By definition,  $N$  is prime if it has no divisors except itself and 1. If  $N$  did have a divisor, it would have to have two, and one of them would have to be less than or equal to  $\sqrt{N}$ . Also, if  $N$  did have a divisor,  $D$ , either  $D$  would be a prime number or  $D$  itself would have at least two other divisors smaller than itself. Thus, we can show that  $N$  is a prime by showing that it is not divisible by any prime less than or equal to  $\sqrt{N}$ .
- If the `divisible()` function returns 1 (true), `k` is not a prime. If it returns 0 (false), we put `k` into the table and increment `n`.

**Last box: printing the table.** The output is printed by calling `print_table()`. The first and last portions of it are

A Table of the First 3000 Prime Numbers

```

2
3
5
7
9
11
13
15
17
19
.....
5993
5995
5997
5999
-----
```

**Notes on Figure 10.17. Functions for the prime number program.** The `divisible()` function can identify primes up to the square of the largest prime currently stored in the table. Its parameters are `candidate` (a number to test), `primes` (a table of primes), and `n` (the current length of the table).

---

These functions are called from Figure 10.16.

```

// -----
// Test candidate number for divisibility by primes in the table.
// Return true if a proper divisor is found, false otherwise.
int
divisible( int primes[], int n, int candidate ) {
    int last = (int) sqrt( candidate );
    int found = 0;           // Initially false, no divisor has been found.

    // Divide by every prime < square root of candidate.
    for (int m = 0; m < n && primes[m] <= last; ++m) {
        if (candidate % primes[m] == 0) {
            found = 1;      // Set to true; divisor has been found.
            break;
        }
    }

    return found;
}

// -----
// Print the list of prime numbers, one per line.
void
print_table( int primes[], int num_primes )
{
    for (int m = 0; m < num_primes; m++) printf( "%10i\n", primes[m] );
    printf( " ----- \n" );
}

```

---

**Figure 10.17. Functions for the prime number program.**

***First and third boxes: the termination condition.***

- We define a variable of type `int`, whose value will be returned later as the result of the function. We initialize the variable to 0 (false) and later set it to 1 (true) if we find what we are searching for; that is, a number that evenly divides the candidate.
- We return the value of `found` after the search either succeeds or exhausts the data in the table.
- This is a common control pattern and especially useful when several tests must be made and any one of them could terminate processing.

***Second box: the search loop.***

- We use the modulus operator to test whether one number is divisible by another;  $a$  is divisible by  $b$  if the remainder of  $a/b$  is 0; that is, if  $a \% b == 0$ .
- To test a candidate number, we divide it by all the numbers in the table up to the square root of the candidate and leave the search loop with a `break` statement the first time we find a proper divisor.
- If no divisor is found, one of two conditions will terminate the loop: Either we have tested every prime in the table or the next prime in the table is greater than the square root of the candidate.

## 10.6 Searching an Array

A common application of arrays is to store a table of data that will be searched, and possibly updated repeatedly, in response to user inputs. In the noncomputer world, a table has at least two columns: a column of index values and one or more columns of data. For example, in a periodic table of the elements, the atomic numbers (1...109) are used as the **index column**, then the element names, atomic weights and chemical symbols are **data columns**. If we use the same data for other purposes, a different column, such as the name, might be chosen as the index column.

A table can either be sorted or unsorted. The data in a **sorted table** are arranged in ascending or descending order, according to some comparison function defined on the values in the index column. For example, a periodic table is sorted in ascending order by the atomic number. A dictionary is sorted in ascending alphabetic order. The typical university course catalog is sorted in ascending order by department code, and within a department, by course number.

To implement a table in the computer, we can use either a set of parallel arrays or an **array of structures**<sup>5</sup>. When we implement a table as a set of **parallel arrays**, we use one array to represent the index column and one more for each data column in the table.

As discussed in Chapter 6, a typical **search loop** examines a set of possibilities, looking for one that matches a given key value. A sequential search of a table examines the index column for an entry that matches the key, one item after another. In every table-searching application, we find the following elements:

- A table, consisting of an index column and one or more data columns.
- A **search key**, the input value that must be compared to the entries in the index column of the table.
- A **comparison function** that is defined for the base type of the array. With simple types, such as numbers or characters, the == operator is appropriate. However, a programmer must define some other comparison function to search an array whose base type is an aggregate type such as those defined in Chapters 12 and 13.
- The **position variable**, the output from the search process, set to the subscript that identifies the value in the index column matching the key value.
- A **success or failure code**, sometimes a separate output value, other times failure may be indicated by setting the position variable to a value either too large or too small to be a legal subscript.

The next program example shows a search loop used for a simple application: recording bill payments in an array of account information. Figure 10.18, gives the specification, Figure 10.19 is the main program, and Figure 10.21 is the **sequential search** function implemented by a search loop.<sup>6</sup>

### Notes on Figure 10.19. Main program for sequential search.

**First box: prototypes for this application.** The main program will call these three functions to do all the work. The first two are more or less the same in every array application that works on parallel arrays: an input function that fills the arrays and an output function that prints them.

**Second box: modeling a table.** We use a parallel-array data structure to implement a table. It has an index column (**ID**) and one data column (**owes**). The maximum capacity of the table is defined at the top of the program, (20 in this case) and the actual number of rows in the table will be determined at run time and stored in **n**.

**Third box: variables for the search.** A sequential search function tries to locate a key value in an array. If it is located, **where** will be used to store its position.

**Fourth box and last box: input, echo, and final output.** We call functions to read the input. In a realistic program, the data would be input from a file rather than from the keyboard. With only minor changes, the code in **get\_data** can be changed to read the data from a file.

The output function is called twice: once to echo the input and again to print the results. When the output code is written in a function, it is easy to use it more than once. This can be especially useful during debugging, when one might wish to see the data in the array within the loop after every change.

<sup>5</sup>Structures are explained in Chapter 13. If a table is modeled as an array of structures, the structure has one member for each column in the table. The array of structures usually is considered a better style because it is more coherent; that is, it groups together all the values for a table entry. The relative merits of these two approaches are discussed in Chapter 13.

<sup>6</sup>Discussion of the binary search algorithm, which is more complex but much faster for sorted arrays, is deferred until recursion is introduced in Chapter 19.

**Problem scope:** Starting with a list of payments that are due, record payment amounts and print a list of account balances after recording the payments.

**Inputs:** Input will happen in two phases.

Phase 1. For each account, enter the account ID number and the initial amount due.

Phase 2. The payments will be entered. For each one, the ID number is entered first. If it is found in the list of accounts, the user will be prompted for a payment amount.

**Constants:** ACCOUNTS must be defined as the maximum number of accounts that the company has simultaneously. The actual number of accounts can be smaller than this limit.

**Output:** For each account number entered in input phase 2, the position of that account in the list will be displayed. At the end of Phase 2, a list of final balances will be displayed. If the bill was overpaid, this balance will be negative.

**Formulas:** Each payment amount should be subtracted from the initial balance in the account.

**Limitations:** No attempt is made to validate payment amounts. ID numbers that are not in the list of accounts will cause an error comment but otherwise have no effect.

**Figure 10.18. Problem specifications: Recording bill payments.**

**Sample output up to this stage.** The first block of output came from `get_data()`, the second block from `print_data()`.

Enter pairs of ID # and unpaid bill (two zeros to end):

```
31      2.35
7       3.19
6       2.28
13      1.09
22      8.83
38      13.25
19      5.44
32      6.90
25      1.70
3       41.
0       0
```

Initial List of Unpaid Bills:

ID	Amount	
[ 0]	31	2.35
[ 1]	7	3.19
[ 2]	6	2.28
[ 3]	13	1.09
[ 4]	22	8.83
[ 5]	38	13.25
[ 6]	19	5.44
[ 7]	32	6.90
[ 8]	25	1.70
[ 9]	3	41.00

10 items were read; ready for payments.

**Fifth box: payments and account balances.** We have read an ID number and are ready to process a payment from that person. First, we must find the position of the person in the table; the call on `sequential_search()` does this, and stores the answer in `where`.

If the ID is not found in the table, `where` will have a negative value. Otherwise, its value will be between 0 and `n-1`. We check this condition, and go on to input and process a payment if the search was successful. Because this is a parallel-array data structure, the payment amount is subtracted from the bill at position `where` in the `owes` array which corresponds to the person at position `where` in the ID array.

*Sample output from this phase.*

```
Enter ID number (zero to end): 31
ID 31  found in slot 0.  Amount paid: 2.35
Enter ID number (zero to end): 25
ID 25  found in slot 9.  Amount paid: 2
```

---

```
#include <stdio.h>
#define ACCOUNTS 20

int get_data( int ID[], float owes[], int nmax );
void print_data( int ID[], float owes[], int n );
int sequential_search( int ID[], int n, int key );

int main( void )
{
    int n;                      // #of ID items; will be <=ACCOUNTS.
    int ID[ ACCOUNTS ];         // ID's of members with overdue bills.
    float owes[ ACCOUNTS ];    // Amount due for each member.

    int key;                    // ID number to search for.
    int where;                  // Position in which key was found.

    float payment;              // Input: amount of payment for one ID#.

    n = get_data( ID, owes, ACCOUNTS );    // Input all unpaid bills.
    printf( "\nInitial List of Unpaid Bills:\n      ID  Amount\n" );
    print_data( ID, owes, n );             // Echo the input

    printf( "\n%i items were read; ready for payments.\n\n", n );
    for (;;) {
        printf( "Enter ID number (zero to end): " );
        scanf( "%i", &key );
        if (key==0) break;

        where = sequential_search( ID, n, key );
        if (where<0) printf( " Item %i \t not found.", key );
        else {
            printf( "\tID %2i \t found in slot %i. ", key, where );
            printf( " Amount paid: " );
            scanf( "%g", &payment );
            owes[where] -= payment;
        }
    }

    printf( "\n\nFinal Amounts Due:\n      ID  Amount\n" );
    print_data( ID, owes, n );           // Print final amounts owed.

    return 0;
}
```

Figure 10.19. Main program for sequential search.

---

```

// -----
int                               // Actual number of data sets read.
get_data( int ID[], float owes[], int nmax )
{
    int k;                      // Loop counter and array index

    printf( "Enter pairs of ID # and unpaid bill (two zeros to end):\n" );
    for (k=0; k<nmax; ++k) {           // Don't go beyond end of arrays.
        scanf( "%i%g", &ID[k], &owes[k] );
        if( ID[k]==0 ) break;          // No more data is available.
    }
    return k;
}
// -----
void print_data( int ID[], float owes[], int n )
{
    for (int k=0; k<n; ++k) {       // Don't read beyond end of ID array.
        printf( "[%2i] %2i %.2f{\bk}\n", k, ID[k], owes[k] );
    }
}

```

---

Figure 10.20. Input and output functions for parallel arrays.

```

Enter ID number (zero to end): 13
ID 13  found in slot 3.  Amount paid: 1
Enter ID number (zero to end): 38
ID 38  found in slot 5.  Amount paid: 10
Enter ID number (zero to end): 0

```

#### Final Amounts Due:

ID	Amount
[ 0]	31 0.00
[ 1]	7 3.19
[ 2]	6 2.28
[ 3]	13 0.09
[ 4]	22 8.83
[ 5]	38 3.25
[ 6]	19 5.44
[ 7]	32 6.90
[ 8]	3 41.00
[ 9]	25 -0.30

**Termination.** Both the `get_data()` function and the payment-processing loop terminate when a sentinel value (an ID number of 0) is entered.

#### Notes on Figure 10.20. Input and output functions for parallel arrays.

**Sequential array processing.** These two functions and the one in Figure 10.21 follow a common pattern used for processing an array sequentially. Each function uses a `for` loop to perform an operation (I/O or calculation) on every array element, starting with the first and ending with the last. A programmer can use arrays for a wide variety of applications by following this pattern and varying the operation.

#### *The get\_data() function.*

- We use variable `k` with a `for` loop to process the array. We must declare `k` before the loop (not inside the `for`'s parentheses) because the value of `k` is used after the end of the loop.

---

A simple sequential search function for an unsorted table.

```
int sequential_search( int ID[], int n, int key )
{
    int cursor; // Loop counter and array index

    for (cursor = 0; cursor < n; ++cursor) {
        if ( ID[cursor] == key ) return cursor;
    }

    return -1;
}
```

---

**Figure 10.21. Sequential search of a table.**

- Before entering the loop, we prompt the user to enter a series of data values. Within the loop, we use a very short prompt to ask the user individually for each value. This is a clear and convenient interactive user interface. During execution of `get_data()`, the user will see something like this:

```
Please enter data values when prompted.
x[0] = 77.89
x[1] = 76.55
x[2] = 76.32
x[3] = 79.43
x[4] = 75.12
x[5] = 64.78
x[6] = 79.06
x[7] = 78.58
x[8] = 75.49
x[9] = 74.78
```

- If we were reading data from a file, an interactive prompt would not be necessary.
- The variable `k` is used as the counter for the loop and also to subscript the array (the usual paradigm for processing arrays). We initialize `k` to 0 and leave the loop when `k` exceeds the last valid subscript, based on the current number of array slots that are in use.
- We also leave the loop if the user enters the sentinel signal: a zero ID number. Because we are reading the ID and the amount owed with the same `scanf()`, a second number must be entered after the sentinel value to “satisfy” the format. Our instructions say to enter two zeros, but the code does not test the second number.
- On each repetition of the loop, we read one data value directly into `&x[k]`, the `k`th slot of the array `x`. At the end of each repetition, we increment `k` to prepare for processing the next array element. Each time around the loop the variable `k` contains a different value between 0 and `n-1`; after `n` iterations, data fill the first `n` array slots and the remaining slots still contain garbage.
- After the sentinel value has been read, we exit from the loop. At this time, the value of `k` is the number of real data items that have been read and stored, excluding the sentinel. We return this essential information to the caller, which will store it and use it to control all future processing on these arrays.
- When control returns to the caller, it can use the values stored in the array by the function.

**Notes on Figures 10.21 and 10.22. Sequential search.** This function assumes that the data are unsorted and that all items must be checked before we can conclude that the search has failed. Two versions are given, a simpler one with two return statements and a longer one with a status flag.

**The function header.** The parameters include elements required for a search algorithm: a table, the number of items to be searched, and a search key. The table is a simple integer array containing `n` values. The return value will be a failure code or the subscript of the key value in the array if it exists.

This is an alternative way to code a search function that uses only one return statement. To accomplish this goal, we use a status flag.

```
int sequential_search( int ID[], int n, int key )
{
    int cursor;      // Loop counter and array index
    int found = 0;   // False now, becomes true if key is found.

    for (cursor = 0; !found && cursor < n; ++cursor) {
        if ( ID[cursor] == key ) found = 1; // true;
    }

    return (found ? cursor-1 : -1);
}
```

**Figure 10.22.** Searching without a second return statement.

In the bill-payment application we use a pair of parallel arrays containing a list of account numbers and the amount owed on each account. Most actions taken by the program (input, output) involve both arrays. However, the arrays are treated differently when we search. Only the index column, in this case the ID array, is passed to the search function.

**Style.** Some experts believe that programmers should never use two return statements in a function. This slightly longer version of the search loop does the same job by using a status flag in place of the second return statement.

**First box: the status flag.** To avoid using either a `break` statement or a second `return` statement, we introduce a status flag named `found`. This is initialized to false (0) and will be set to true (1) within the search loop if the key item is found.

**Second box: the search loop.** A counted loop is used to examine the data items. If a match is found for the key, the loop terminates; otherwise, all `n` values are checked.

**First inner box: the comparison.** Since the base type of the array is `int`, we use the `==` operator to compare each item to the key value. The search succeeds if the result is true (1).

**Second inner box: Dealing with Success.** If a match is found, we need to leave the loop. This is done in different ways by our two versions of this function.

- Sometimes we abort a loop with a `break` statement. Here, we use `return` for the same purpose. It causes control to leave both the loop and the function. The current item's subscript is returned to the caller.
- We avoid using a second return statement tests by setting a status flag to true (1), indicating that the key value has been found. Control does not leave the loop. It returns to the top of the loop and increments `cursor`, making it one too large. The function return statement will have to compensate for this extra increment.

**Last box, Figure 10.21: Failure.** If the loop goes past the last data item without finding a match, the search has failed. Failure is indicated by returning a subscript of `-1`, a subscript that is invalid for any array, and often used to indicate error or failure.

**Last box, Figure 10.22: Returning.** This single return statement must handle both success and failure. If the item was not found, we want to return `-1` to indicate failure. If it was found, we must return the value of `cursor-1`. The value of `cursor` is too large by 1 because the `for` loop incremented it after `found` was set to true and before `found` was tested to terminate the loop.

The little-used conditional operator provides a way to use a single return statement to return one thing or another. It works like an `if...else` except that it is an expression, not a statement. Read the statement like this: "If `found` is true, then return `cursor-1` else return `-1`".

```
#include <stdio.h>
#define ACCOUNTS 20

int get_data( int ID[], float owes[], int nmax );
int find_max( int ID[], int n );

int main( void )
{
    int n;                      // #of ID items; will be <=ACCOUNTS.
    int ID[ ACCOUNTS ];         // ID's of members with overdue bills.
    float owes[ ACCOUNTS ];    // Amount due for each member.
    int where;                  // Position of maximum value.

    n = get_data( ID, owes, ACCOUNTS ); // Input all unpaid bills.
    printf( "\nLargest Unpaid Account:\n" );

    where = find_max( owes, n );
    printf( "\tID# %i owes $ %.2f\n", ID[where], owes[where] );

    return 0;
}
```

Figure 10.23. Who owes the most? Main program for finding the maximum.

## 10.7 The Maximum Value in an Array

Finding the maximum value in an array is an easy but nontrivial task. The `find_max()` function presented here scans the data array sequentially, like a search function, but it does not use a search key. To illustrate this algorithm, we use the same parallel-array data structure that was used in Figure 10.19, and search the data for the largest unpaid bill.

### Notes on Figure 10.23. Who owes the most?

**First box: Prototypes.** The `get_data()` function is in Figure 10.21 and `find_max` is in Figure 10.24.

**Second box: data declarations.** We are using the same data structure as in the sequential search application: a set of parallel arrays containing the ID numbers and unpaid balances of a set of up to `ACCOUNTS` customers. We need `n` to store the actual number of customers that were input and `where` to store the position of the customer with the largest bill.

```
int                                         // Find the maximum value in an array.
find_max( float data[], int n )
{
    int finger = 0;                      // Put your finger on the first value.
    int cursor;

    for (cursor = 1; cursor < n; ++cursor) {
        if (data[finger] < data[cursor]) // If you find a bigger value...
            finger = cursor;           // ...move your finger to it.
    }

    return finger;                      // Your finger is on the biggest value.
}
```

Figure 10.24. Finding the maximum value.

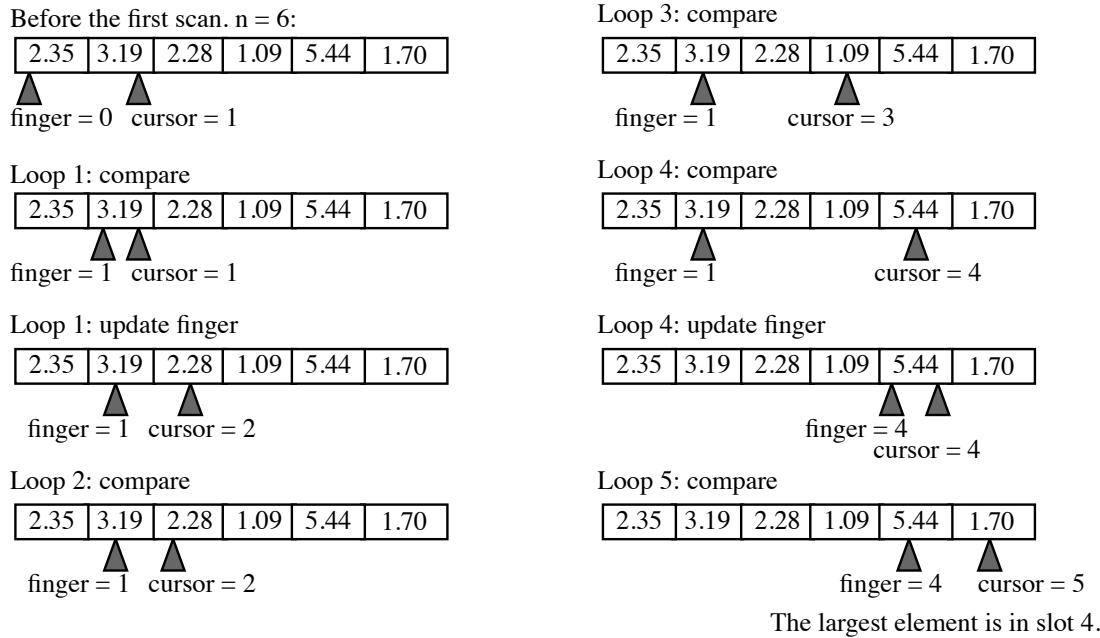


Figure 10.25. The maximum algorithm, step by step.

**Third box: doing the work.** We call the `find_max()` function in Figure 10.24 to search for the largest value in the array `owes` which contains `n` data items. The result is stored in `where`, and is then used to print out both the biggest bill and the ID number of the customer who owes the most.

#### Notes on Figure 10.24. Finding the maximum value.

**The function header.** The `find_max()` function is called from `main()` in Figure 10.23. We only need two parameters: the array to search and the length of that array.

**First box: initialization.** The idea is to scan the array sequentially, but at all times, to keep one “finger” on the biggest value we have seen so far. To get started, we set `finger` to slot 0.

**Second box: the search loop.** To find the largest value in an array, we must examine every array element. We start by designating the first value as the biggest-so-far, then scan start a sequential scan from array slot 1, looking for something bigger. Whenever we find the value under the `cursor` is bigger than the one under the `finger`, we update `finger`. When the loop ends, `finger` will be the position of the largest value. This process is illustrated in Figure 10.25.

## 10.8 Sorting by Selection

A common operation performed on arrays is sorting the data they contain. Many sorting methods have been invented: Some are simple, some complex, some efficient, some miserably inefficient. In general, the more complex sorting algorithms are the most efficient, especially if the array is very long.

In this section, we look at **selection sort**, which can be used on a small number of items. It is one of the simplest sorting methods, but also one of the slowest. Nonetheless, selection sort has the advantage that, if you stop in the middle of the process, one part of the array is fully sorted, so it is a reasonable way to find the either the largest or smallest few items in a long array.<sup>7</sup>

<sup>7</sup>A simple algorithm, **insertion sort**, has been shown to be the fastest sort of all when used on short arrays (fewer than 10 items). We present this algorithm in Chapter 16. The **quicksort** (discussed in Chapter 19) is a much better way to sort

The basic selection strategy has several variations: the data can be sorted in ascending or descending order, the work can be done by using either a maximum or a minimum function, and the sorted elements can be collected at either the beginning or the end of the array. In this section, we develop a version that sorts the array elements in ascending order, by using a maximum function and collecting the sorted values at the end of the array. At any time, the array consists of an unsorted portion on the left (initially the whole array) and a sorted portion on the right (initially empty). To sort the array, we make repeated trips through smaller and smaller portions of it. On each trip, we locate the largest remaining value in the unsorted part of the array, then move it to the beginning of the sorted area by swapping it with whatever value happens to be there. After moderate length and long arrays. Two other simple sorts, bubble sort and exchange sort, have truly bad performance for all applications. They are not presented here and should not be used.

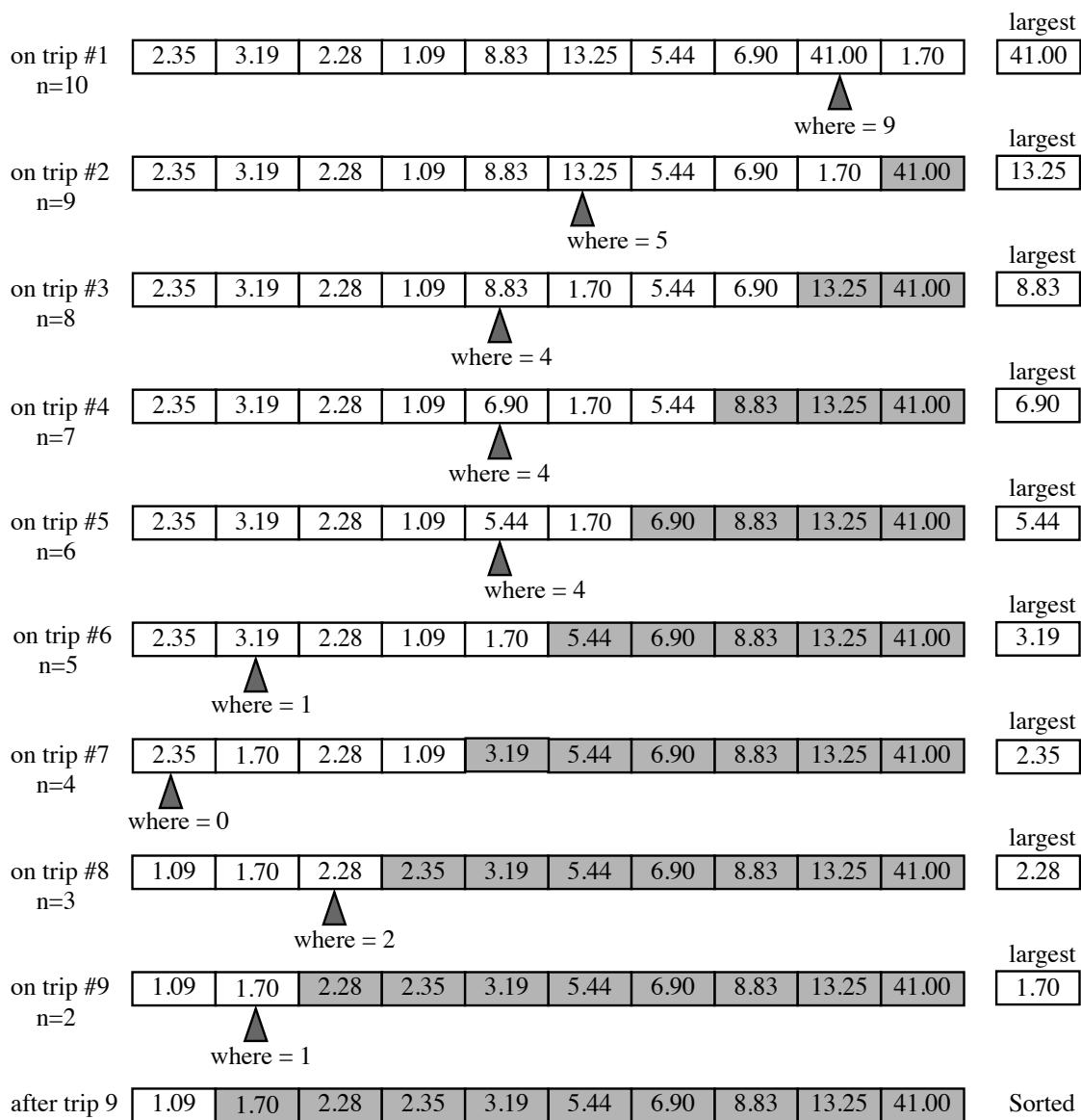


Figure 10.26. The selection sort algorithm, step by step.

**Goal:** Sort customer billing records in ascending order according to the amount owed. These records are stored in a parallel-array data structure with two columns: ID number and amount owed. There are  $n$  records altogether.

**Input:** ID numbers and amounts due for a set of customers.

**Output:** A list of customer records, in order, by the amount owed.

**Algorithm:** Use a selection sort, as follows:

Repeat the following actions  $n - 1$  times:

1. Let **where** be the subscript of the maximum-valued element in the array between subscripts 0 and  $n-1$ .
2. Swap the element at position **where** with the element at position  $n-1$ . The swapped value will now be in its proper sorted position.
3. Decrement **n** to indicate that there are now fewer unsorted items.

---

**Figure 10.27. Problem specifications: Sorting the Billing Records**

$k$  trips, the  $k$  largest items have been selected and placed, in order, at the end of the array. After  $n - 1$  trips, the array is sorted. This process is illustrated in Figure 10.26.

A program to implement this algorithm is developed easily by a top-down analysis. A problem specification is given in Figure 10.27.

### 10.8.1 The Main Program

A well-designed main program is like an outline of the process; it calls on a series of functions to do each phase of the actual job. This kind of design is easy to plan, easy to read, and easy to debug. The sorting task has three major phases: input (read the numbers), processing (sort them), and output (print the sorted list). For each phase, the main program should call a function to do the job and display a comment that reports the progress. Programs that contain arrays and loops often take a while to debug; during that time, generous feedback helps the programmer identify the location and nature of the errors.

We start by writing the obvious parts of `main()`, borrowing elements from the previous programs (sequential search and finding the maximum), where possible. Much can be borrowed, including

- Prototypes for the I/O functions that work with the parallel arrays (`get_data()` and `print_data()`),
- The skeleton of `main()`,
- Declarations within `main()` for the parallel-array data structure.
- Since the selection sort algorithm must find the maximum value in an array, we also include the prototype for `find_max()`.

#### Step 1: The obvious necessities.

```
#include <stdio.h>
#define ACCOUNTS 20

int get_data( int ID[], float owes[], int nmax );
print_data( ID, owes, n );           // Print final amounts owed.
find_max( int ID[], int n );        // Located largest key value in array.

int main( void )
{
    int n;                         // #of ID items; will be <=ACCOUNTS.
    int ID[ ACCOUNTS ];            // ID's of members with overdue bills.
    float owes[ ACCOUNTS ];        // Amount due for each member.
    ...
    return 0;
```

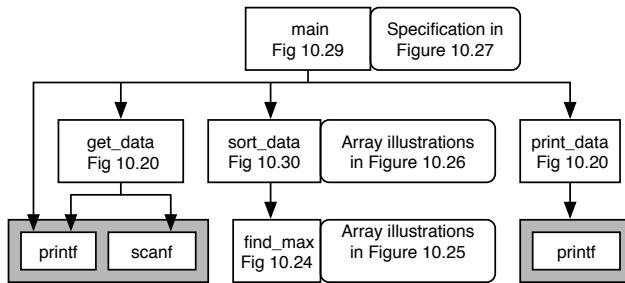


Figure 10.28. Call chart for selection sort.

}

**Step 2: Input.** At the position of the dots, we add calls on the input and output functions, following the example of Figure 10.19.

```

n = get_data( ID, owes, ACCOUNTS ); // Input all unpaid bills.
printf( "\nInitial List of Unpaid Bills:\n      ID  Amount\n" );
print_data( ID, owes, n );           // Echo the input

```

**Step 3: Processing.** We invent the name `sort_data()` for the selection sort function. In general, the function needs to know what array to sort and the length of that array. It rearranges the data within the array and returns the sorted values in the same array, with no need for any additional return value. In this case, we are sorting a pair of parallel arrays, so both arrays must be rearranged in parallel. Thus, the `sort_data()` function must take both arrays as parameters. We write a prototype (at the top) and a call for this function inside `main()`:

```

void sort_data( int data[], float key[], int n );
...
sort_data( ID, owes, n );

```

**Output.** When sorting is finished, we need to output the sorted data. So we add another call on `print_data()` at the end of `main()`:

```

print_data( ID, owes, n );           // The sorted data

```

A call chart for the overall program is given in Figure 10.28. The completed `main()` function is shown in Figure 10.29.

### 10.8.2 Developing the `sort_data()` Function

We must start with a thorough understanding of the algorithm. This is illustrated in Figures 10.25 and 10.26 and defined carefully in Figure 10.27. Once the method is understood, we are ready to implement the `sort_data()` function. (The code fragments that follow are assembled in Figure 10.30.) We begin with the function skeleton and declare the variables mentioned in the problem specifications.

```

void sort_data( int data[], float key[], int n );
{
    int where; ...
}

```

**The loop skeleton and body.** Step 2 of the specification calls for a process to be repeated  $n - 1$  times to sort  $n$  things. We write a `for` loop that implements this control pattern:

---

This program calls the sort function in Figure 10.30, and the input and output functions from Figure 10.21.

```
#include <stdio.h>
#define ACCOUNTS 20

int get_data( int ID[], float owes[], int nmax );
void print_data( int ID[], float owes[], int n );
void sort_data( int data[], float key[], int n );

int main( void )
{
    int n;                      // #of ID items; will be <=ACCOUNTS.
    int ID[ ACCOUNTS ];         // ID's of members with overdue bills.
    float owes[ ACCOUNTS ];    // Amount due for each member.

    n = get_data( ID, owes, ACCOUNTS );   // Input all unpaid bills.
    printf( "%i items were read; beginning to sort.\n", n );
    sort_data( ID, owes, n );

    puts( "\nData sorted, ready for output" );
    print_data( ID, owes, n );
    return 0;
}
```

---

**Figure 10.29.** Main program for selection sort.

```
for (start=n-1; start>0; --start) {
    ...
}
```

Now we need to write the body of the loop. We have a function that finds the maximum value in an array, so we call it and save the result.

```
where = find_max( key, n );
```

Next, we must swap the large value at position `where` with the last unsorted value in the array, which is at position `start`. Swapping takes three assignments, but since we are working on a parallel-array data structure, identical swaps must be made on both arrays, for a total of six assignments. In the code below, the instructions on the left swap the key array, those on the right swap the data array.

Also, on each repetition, we start at the high-subscript end of the unsorted array, that is, at slot  $n - 1$ . On each repetition, one item is placed in its final position, leaving one fewer item to be sorted. So we decrement  $n$  just before the end of the loop.

```
bigKey = key[where];
key[where] = key[start];
key[start] = bigKey;
--n;
```

This completes the algorithm and the program. We gather all the parts of `sort_data()` together in Figure 10.30.

**Testing.** We combined all of the pieces of the sort program, compiled it, and ran it on the file `sele.in`, which contains the data listed in Figure 10.31. The input is on the left and the corresponding output on the right.

---

This function is called from `main()` in Figure 10.29; it calls `find_max()` in Figure 10.24.

```
int find_max( float data[], int n ); // Prototype not included by main().

void sort_data( int data[], float key[], int n )
{
    int start;      // End of unsorted data in the array.
    int where;      // Position of largest value in the key array.
    int bigData;    // For swapping the data column.
    float bigValue; // For swapping the key column.

    for (start=n-1; start>0; --start) {
        where = find_max( key, n );
        // Swap the two columns of the table in parallel.
        bigValue = key[where];      bigData = data[where];
        key[where] = key[start];   data[where] = data[start];
        key[start] = bigValue;     data[start] = bigData;
        --n;
    }
}
```

---

**Figure 10.30.** Sorting by selecting the maximum.

## 10.9 What You Should Remember

### 10.9.1 Major Concepts

**Arrays and their use.** An array in C is a collection of variables stored in order, in consecutive locations in the computer's memory. The array element with the smallest subscript (0) is stored in the location with the lowest address. We use arrays to store large collections of data of the same type. This is essential in three situations:

- When the individual data items must be used in a random order, as with the items in a list or data in a table.
- When each data value represents one part of a compound data object, such as a vector, that will be used repeatedly in calculations.

Input Phase	Output Phase
Enter pairs of ID and unpaid bill (two zeros to end): 31            2.35 7            3.19 6            2.28 13            1.09 22            8.83 38            13.25 19            5.44 32            6.90 3            41. 25            1.70 0            0	Data sorted, ready for output [ 0] 13    1.09 [ 1] 25    1.70 [ 2] 6    2.28 [ 3] 31    2.35 [ 4] 7    3.19 [ 5] 19    5.44 [ 6] 32    6.90 [ 7] 22    8.83 [ 8] 38    13.25 [ 9] 3    41.00

---

**Figure 10.31.** Input and output for selection sort.

- When the data must be processed in separate phases, as in the problem from Figure 10.18. In this program, all the account balances must first be read and stored. Then the stored data must be searched and updated, using bill-payment amounts.

**Parallel arrays.** A multicolumn table can be represented as a set of parallel arrays, one array per column, all having the same length and accessed using the same subscript variable. Multidimensional arrays also exist and are discussed in Chapter 18.

**Array arguments and parameters.** An array name followed by a subscript in square brackets denotes one array element whose type is the base type of the array. This element can be used as an argument to a function that has a parameter of the base type. To pass an entire array as an argument, write just the array name with no subscript brackets. The corresponding formal parameter is declared with empty square brackets. When the function call is executed, the address of the beginning of the array will be passed to the function. This gives the function full access to the array; it can use the data in it or store new data there.

**Array initializers.** C allows great flexibility in writing array initializers; we summarize the rules here:

- An array initializer is a series of constant expressions enclosed in curly brackets. These expressions can involve operators, but they must not depend on input or run-time values of variables. The compiler must be able to evaluate the expressions at compile time.
- If there are too many initial values for the declared length, C will give a compile-time error comment.
- If there are too few initial values, the uninitialized areas will be filled with 0 values of the proper type: an integer 0, floating value 0.0, or pointer NULL.
- The length of an array may be omitted from the declaration if an initializer is given. In this case, the items in the initializer will be counted and the count will be used as the length of the array.

**Searching.** Many methods can be used to search an array for a particular item or one with certain characteristics. A sequential search starts at the beginning of a table and compares a key value, in turn, to every element in the key column. The search ends when the key item is found or after each item has been examined.

The typical control structure for implementing a sequential search is a `for` loop that moves a subscript from the beginning of the array to the end. In the body of this loop is an `if...break` statement that compares the key value to the current table element.

The search can either succeed (find the key value) and break out of the loop or fail (because the key value does not match any item in the table). A sequential search for a specific item is slow and appropriate for only short tables. It is slightly more efficient when the table is in sorted order, because failure can be detected prior to reaching the end of the table. However, binary search (see Chapter 19) is an even faster algorithm for use with sorted data.

If the data are sorted according to the search criterion, shortcuts may be possible. However, a sequential search is necessary when the order of the data in the array is unrelated to the criterion because all the data items must be examined. For example, finding the longest word in a dictionary would require looking at every word (a sequential search) because a dictionary is sorted alphabetically, not by word length.

**Sorting.** Locating a particular item in a table can be done much more efficiently if the information is sorted. Many sort algorithms have been devised and studied; among the simplest (and slowest) is the selection sort. It sorts  $n$  items by selecting the minimum remaining element  $n - 1$  times and moving it to a part of the array that will not be searched again. Other more efficient techniques such as the insertion sort (Chapter 16) and the quicksort (Chapter 19) are examined later.

## 10.9.2 Programming Style

### *Usage.*

- Use a defined constant to declare the length of an array. This way the use and the array declarations will be consistent and easily changed if the need arises.
- A `for` loop typically is used to process the elements of an array. The values of the loop counter go from 0 to the length of the array (which is given by a defined constant or a function parameter); for example, `for (k=0; k<N; ++k) printf( "%g ", volume[k] );`. Note that this loop paradigm stops before attempting to process the nonexistent element `volume[N]`.

**Names.** Variable names such as `j` and `k` typically are used as array subscripts since they are commonly found in mathematical formulas. However, when writing a program that sorts, it is very helpful to use meaningful names for the subscript variables. You are much more likely to write the code correctly in the first place, and then get it debugged, if you use names like `cursor` and `finger` rather than single-letter variable names such as `i` and `j`.

**Local vs. global.** Constants should be declared globally if they are used by more than one function or if they are purely arbitrary and likely to be changed. If a constant is used by only one function, it may be better to declare it locally. However, a large set or table of such constants will incur large setup times each time the function is called. These constants should be declared as `static const` values, which are only initialized once.

**Don't talk to strangers.** Each object name used in a function should represent an object that fits into one of the following categories:

- A global constant, `#defined` at the top of the program, or like `NULL`, in a header file.
- A parameter, declared and named in the function header.
- A local variable or constant, declared and named within the function (an object should be local if it is used only within a function and does not carry information from one function to another).
- A global function whose prototype is at the top of the program or in a header file.
- A local function, declared within the function and defined after it (a function should be declared locally if it is used only within that function. This does not cause any run-time inefficiency).

**Modularity.** We wrote a `sort_data()` function as a loop that calls the `find_min()` function. Within that function is another loop. When written like this, the logic of the program is completely transparent and easily understood. In many texts, this algorithm is written as a loop within a loop. This second form takes fewer lines of code and executes more efficiently, because no time is spent calling functions. However, it is not so easy to understand. Which form is better? The modular form. Why? Because it can be debugged more easily and is less likely to have persistent bugs. Doesn't efficiency matter? It often does, but if so, a better algorithm (such as quicksort) should be used instead. It is a false economy to use bad programming style to optimize a slow algorithm.

**Sorted vs. unsorted.** If the data we wish to search already are sorted, by all means we should take advantage of this. If not, we need to decide whether to sort the data before searching. This issue will be addressed to some extent in later chapters. However, it is a complex issue involving the data set size, how fast the data set changes, which data structures and algorithms are used, and how many times a search will be performed. The general topic of data organization and retrieval is the subject of dozens of books on data structures and databases.

**Software reuse.** Do not waste time trying to reinvent the wheel. If a library routine meets your need, use it. If you have previously written a function that does almost what you need, modify it as necessary. If someone else has developed a solution for a certain task, such as sorting, go ahead and use it, after you have verified that any assumptions it makes are satisfied by your data and structures.

### 10.9.3 Sticky Points and Common Errors

**Array length vs. highest subscript.** The number given in an array declaration is the actual number of slots in the array. Since array subscripts start at 0, the highest valid subscript is one less than the declared length. Often, though, there are more slots than valid data. This happens during an input operation and whenever the total amount of data entered falls short of the maximum allowed. In such situations, another variable is used to store the number of actual data elements in the array.

**Subscript errors.** Programmers accustomed to other languages often are surprised to learn that C does absolutely no subscript range checking. If a subscript outside the defined range is used, there will be no error comment from the compiler or at run-time. The program will run and simply access a memory location that belongs to some other variable. For example, if we write a loop to print the values in an array and it loops too many times, the program starts printing the values adjacent to the array in memory. At best, this results in minor errors in the results; at worst, the program can crash.

**Caution: do not fall off the end of an array.** Remember that C does not help you confine your processing to the array slots that you defined. When you use arrays, avoid any possibility of using an invalid subscript. Input values must be checked before using them as subscripts. Loops that process arrays must terminate when the loop counter reaches the number of items in the array.

**Ampersand errors.** Arrays and nonarrays are treated differently in C. An array argument always is passed to a function by address. We do not need to use an ampersand with an unsubscripted array name.

**Array parameters.** To pass an entire array as an argument, write just the name of the array, with no ampersands or subscript brackets. The ampersand operator is not necessary for an array argument because the array name automatically is translated into an address. The corresponding array parameter is declared with the same base type and empty square brackets. A number can be placed between the brackets, but it will be ignored.

**Array elements as parameters.** A single array element also can be passed as a parameter. To do this, write the array name with square brackets and a subscript. If the function is expected to store information in the array slot, as `scanf()` might, you must also use an ampersand in front of the name.

**Sorting.** Writing a sorting algorithm can be a little tricky. It is quite common to write loops that execute one too many or one too few times. When debugging a sort, be sure to examine the output closely. Check that the items at the beginning and end of the original data file are in the sorted file and that the output has the correct number of items. Examine the items carefully and make sure all are there and in order. It is common to make an error involving the first or last item. Test all programs on small data sets that can be thoroughly checked by hand.

**Parallel arrays.** A table can be implemented as a set of parallel arrays. When sorting such a table, it is important to keep the arrays all synchronized. If the items in one column are swapped, be sure to swap the corresponding items in all other columns. Using an array of structures may solve this problem, but this solution may have its own drawbacks, which we have discussed previously.

#### 10.9.4 Where to Find More Information

- Arrays of strings are presented in Chapter 12, arrays of structures in Chapter 13 and arrays of functions in Chapter 16.
- Dynamic allocation of arrays and arrays of pointers are found in Chapter 16.
- Pointers are introduced in Chapter 11 and the use of pointers to process arrays is explained in Chapter 16.
- Two dimensional arrays and their applications are covered in Chapter 18. Multidimensional arrays and arrays of pointers to arrays are in the same chapter.
- Other sorting algorithms are presented in later chapters. Insertion sort is in Chapter 16; quicksort in Chapter 19.
- Other array algorithms presented are: Binary search: Chapter 19, Shuffling a deck: Chapter 14, Gaussian elimination: Chapter 18, Simulation: Chapter 16.

#### 10.9.5 New and Revisited Vocabulary

These are the most important terms and concepts presented in this chapter:

aggregate type	constant expression	status flag
array	parallel arrays	sequential search
base type	effective address	search loop
array slot	walking on memory	key column
array element	memory error	data column
array length	array argument	search key
size of an array	array parameter	sorted table
subscript	sequential array processing	position variable
array declaration	prime number	finding the minimum
array initializer	divisibility	selection sort

## 10.10 Exercises

### 10.10.1 Self-Test Exercises

1. Which occupies more memory space, an array of 15 `short ints` or an array of 3 `doubles`? Explain your answer.
2. An array will be used to store temperature readings at four-hour intervals for one day. It is declared thus: `float temps[6];`
  - (a) Draw an object diagram of this array.
  - (b) What is its base type? Its length? Its size?
  - (c) Write a loop that will read data from the keyboard into this array.
  - (d) Write an `if` statement that will print `freezing` if the temperature in the last slot is less than or equal to  $32^{\circ}\text{F}$  and `above freezing` otherwise.
3. Array and function declarations.

- (a) Write a declaration with an initializer for the array of `floats` pictured here.

ff				
1.9	2.5	-3.1	17.2	0
[0]	[1]	[2]	[3]	[4]

- (b) Write a complete function that takes this array as a parameter, looks at each array element, and returns the number of elements greater than 0.
- (c) Write a prototype for this function.
- (d) Write a `scanf()` statement to enter a value into the last slot of the array.
4. In the indicated spots below, write a prototype, function header, and call for a function named `CHKBAL` that computes and returns the balance in a checking account. Its parameters are an initial account balance and an array of check amounts. You need not actually write a whole function to compute the new account balance; just fill in the indicated information.

```
#define X 5
// insert prototype here

int main( void )
{
    float check_amounts[X];    // $ amounts of checks
    float start_balance;       // balance before checks
    float end_balance;         // balance after checks
    // put call here
}
// put function header here
```

5. The following are two declarations and a `while` loop.

```
int ara[13] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096};
int k = 12;
while (k > 0) {
    printf( "%2i: %i \n", k, ara[k] );
    k -= 2;
}
```

- (a) What is the output?

- (b) Rewrite the code using a `for` loop instead of the `while` loop.
6. Given the following declarations, the prototypes on the left, and the calls on the right, answer *good* if a function call is legal and meaningful. If there is an error in the call, show how you might change it to match the prototype.

```
int j, a[5];
float f, flo ;
double x, dub[4];
```

- |                                    |                        |
|------------------------------------|------------------------|
| (a) double fun( double d[] );      | x = fun( dub[] );      |
| (b) void fill( double d );         | x = fill( dub );       |
| (c) int fix( float f );            | fix( flo[5] );         |
| (d) int hack( int a[] );           | j = hack( a );         |
| (e) double q( double d[], int n ); | x = q( dub[0], a[0] ); |

7. Use the following definitions in this problem:

```
#define LIMIT 5
int j;
double load[LIMIT];
```

- (a) Draw an object diagram for the array named `load`; identify the slot numbers in your drawing. Also show the values stored in it by executing the following loop:
- ```
for ( j=0; j<LIMIT; j++ ) {
    if ( j % 2 == 0 ) load[j] = 10.2 * (j + 1);
    else load[j] = (j + 1) * 2.5;
}
```
- (b) Trace the following loop and show the output exactly as it would appear on your screen. Show your work.
- ```
for (j = 0; j < LIMIT; ++j) {
    if (j <= LIMIT / 2) printf( "\t%6.3g", load[j] );
    else printf( "\t%5.2f", load[j] - .05 );
}
```
8. The following diagram shows an array of odd integers. Declare and initialize a parallel array of type `int` that contains 1 (true) in the slot corresponding to every prime number, and 0 (false) for the nonprime numbers.

3	5	7	9	11	13	15	17	19	21	23	25
---	---	---	---	----	----	----	----	----	----	----	----

9. Consider an array containing six data items that you must sort using the selection sort algorithm. How many data comparisons does the algorithm perform in the `find_min()` function? How many times is `find_min()` called, and how many comparisons are made (total) during all these calls? How many (total) data swaps does `sort_data()` perform?
10. Suppose you are searching for an item in a sorted array of  $N$  items. Using a sequential search algorithm, how many items are you likely to check before you find the right one? Is this number the same whether or not the item is present in the array? Express the answer as a function of  $N$ .
11. The selection sort in the text generated a list of values in ascending order. How would you change the algorithm to generate numbers in descending order?
12. Consider the following list of numbers: 77, 32, 86, 12, 14, 64, 99, 3, 43, 21. Following the example in Figure 10.24, show how the numbers in this list would be sorted after each pass of the selection sort algorithm.

### 10.10.2 Using Pencil and Paper

1. Draw a flow diagram for the main program in Figure 10.16 and for the `divisible()` function in Figure 10.17 (you may omit the details of the other function). In your diagram, show how control flows from one function to the other and back again.

2. An array will be used to store the serial numbers of the printers in the lab. It is declared thus:

```
long printer_ID[10];
```

- (a) Draw an object diagram of this array.
  - (b) What is its base type? Its length? Its size?
  - (c) Write a loop that will read data from the keyboard into this array; the loop should end when the user enters a negative number. Store the actual number of data items in the variable named `count`.
  - (d) Write a loop that will print out all `count` data items from the array.
3. Array and function declarations.

- (a) Write a declaration with an initializer for the array of small integers pictured here:

scores			
96	68	79	93
[0]	[1]	[2]	[3]

- (b) Write a complete function that takes this array and an integer `n` as parameters, tests each array element, and prints all array elements greater than `n`.
- (c) Write a prototype for this function.
4. Use the following definitions in this problem:

```
#define LIMIT 7
int j;
short puzzle[LIMIT], crazy[LIMIT];
```

- (a) Draw an object diagram for the array named `puzzle`; identify the slot numbers in your drawing. Also show the values stored in it by executing the following loop. This loop performs an illegal operation. Your job is to figure out what will happen and why.

```
for (j = LIMIT; j > 0; --j) {
    if (j+4 > j*2) {
        puzzle[j] = j*2;
        crazy[LIMIT-j] = j + 2;
    }
    else {
        puzzle[j] = j/2;
        crazy[LIMIT-j] = j - 2;
    }
}
```

- (b) Trace the following loop and show the output exactly as it would appear on your screen. Show your work.

```
for (j = 0; j < LIMIT; ++j) {
    if (j % 2 == 1)
        printf( "\t%5i %3i", puzzle[j], crazy[j] );
    else printf( "\t%3i %5i", crazy[j], puzzle[j] );
}
```

5. The following are two declarations and a `for` loop.

- (a) What is the output?
- (b) Rewrite the code using a `while` loop instead of the `for` loop.

```
int a[12] = {2, 4, 8, 3, 9, 27, 4, 16, 64, 5, 25, 125};
for (int k = 1; k < 10; k += 2) printf("%i: %i \n", k, a[k]);
```

6. Given the following declarations, the prototypes on the left, and the calls on the right, answer *good* if a function call is legal and meaningful. If there is an error in the call, show how you might change it to match the prototype.

<pre>int j, ary[5]; float f, flo ; double x, dub[4];</pre>	<pre>(a) double list( double d[] ); (b) void handle( int k ); (c) void bank( float f, int n ); (d) int days( int a[] ); (e) int area( double d );</pre>	<pre>x = list( ary ); handle( ary[5] ); bank( flo[5], j ); j = days( &amp;flo ); x = area( dub[0] );</pre>
--	---	--

7. In the spots indicated below, write a prototype, function header, and call for a function named MISSING. Its parameters are an array of student assignment scores and the number of assignments that have been graded and recorded. The function will look at the data and return the number of nonzero scores in the array. You need not actually write the whole function, just fill in the indicated information.

```
#define MAX 10
// insert prototype here

int main( void )
{
    int assignments[MAX]; // grades
    int actual;           // # of assignments so far.
    int done;             // # of nonzero grades.
    // put call here
}

// put function header here
```

8. An unsuccessful search for an item in sorted and unsorted data arrays will require different numbers of comparisons. Compare a sequential search on these two types of data and explain why they are different (in terms of the number of comparisons performed).
9. Modify the code in the sequential search function in Figure 10.21 so that it assumes the data in the array are sorted in descending order. Do not search any more positions than necessary. Still return a value of -1 if the key value cannot be found.
10. Consider an array containing  $N$  data items that you must sort using the selection sort algorithm. How many data comparisons does the algorithm perform? How many data swaps does it perform? Express the answer as a function of  $N$ .
11. Consider the following list of numbers: 77, 32, 86, 12, 14, 64, 99, 3, 43, 21. Following the example in Figure 10.26, show how the numbers in this list would be sorted after each pass of a selection sort algorithm that sorts numbers in *descending* order.

### 10.10.3 Using the Computer

1. Seeing bits.

The program in Figure 4.23 shows how to convert an integer to any selected base. Obviously, this program works for base 2, binary notation. Modify this program so that it inputs a number and prints the equivalent value in binary notation. Do not print it in the expanded form of the previous program, but as a series of ones and zeros. You will need to store the binary digits in an array and print them later in the opposite order, so that the first digit generated is the last printed. For example, if the input were 22, the output should be 10110.

2. Global warming.

As part of a global warming analysis, a research facility tracks outdoor temperatures at the North Pole once a day, at noon, for a year. At the end of each month, these temperatures are entered into the computer and processed. The operator will enter 28, 29, 30, or 31 data items, depending on the month. You may use -500 as a sentinel value after the last temperature, since that is lower than absolute 0. Your main program should call the `read_temps()`, `hot_days()`, and `print_temps()` functions described here:

- (a) Write a complete specification for this program.
- (b) Write a function, `read_temps()`, that has one parameter, an array called `temps`, in which to store the temperatures. Read the real data values for one month and store them into the slots of an array. Return the actual number of temperatures read as the result of the function.
- (c) Write a function, `hot_days()`, that has two parameters: the number of temperatures for the current month and an array in which the temperatures are stored. Search through the temperature array and count all the days on which the noon temperature exceeds 32°F. Return this count.
- (d) Write a function, `print_temps()`, with the same two parameters plus the count of hot days. Print a neat table of temperatures. At the same time, calculate the average temperature for the month and print it at the end of the table, followed by the number of hot days.

3. The tab.

An office with six workers maintains a snack bar managed on the honor system. A worker who takes a snack records his or her ID number and the price on a list. Once a month, the snack bar manager enters the data into a computer program that calculates the monthly bill for each worker. No item at the snack bar costs more than \$2, and monthly totals are usually less than \$100.

- (a) Write a complete specification for this program.
- (b) Using a top-down development technique, write a main program that will call functions to generate a monthly report. These functions are described here. Declare an array of `floats` named `tabs` to store total purchase amounts for each member and the guests.
- (c) The `purchases()` function should have one parameter, the `tabs` array. This function should allow the manager to enter two data items for each purchase: the price and the ID number of the worker who made the purchase. The ID numbers must be integers between 1 and 6. In addition, the code 0 is used for guests, whose bills are paid by the company. As each purchase is read, the amount (in dollars and cents) should be added to the array slot for the appropriate worker. When the manager enters an ID code that is not between 0 and 6, it should be considered a sentinel value and a signal to end the loop and return from the function. At that time, the array should contain the total purchases for each worker and for the guests.
- (d) The `bills()` function should have one parameter, the `tabs` array. Print a bill for each worker, giving the ID number and the amount due.

4. Payroll.

The Acme Company has some unusual payroll practices and keeps the information in its personnel database in a strange way. The firm never has more than 200 employees and pays all its employees twice a month, according to the following rules:

- (a) If the person is salaried, the pay rate will be greater than \$1,000. There are 24 pay periods per year, so for one period, `earnings = payrate / 24`.
  - (b) If the person is paid hourly, the pay rate will be between \$5 and \$100 per hour and the earnings are calculated by this formula:  

$$\text{earnings} = \text{payrate} * \text{hours}$$
  - (c) A pay rate less than \$5 per hour or between \$100 and \$1,000 per hour is invalid and should be rejected.
- (a) Write a complete specification for this program.

- (b) Using a top-down development process and following the example of Figure 10.13, write a main program that prints the bimonthly payroll report. Since the number of employees changes frequently, `main()` should prompt for this information. Then call functions to perform the calculations and produce a report. Use the functions suggested here, and add more if that seems appropriate. You will need a set of parallel arrays to hold the ID number, pay rate, hours worked, and earnings for each employee.
  - (c) Following the example of Figure 10.21, write a function, `get_earnings()`. Read the data for all the employees from the keyboard into the parallel arrays.
  - (d) Write a function, named `earn()`, that uses the pay rate and hours worked arrays to calculate the earnings and fill in the earnings array.
  - (e) Write a function, named `pay()`, to calculate and return the earnings for one employee. If the pay rate is invalid, print an error comment and return 0.0.
  - (f) Write a function, named `payroll()`, that prints a neat table of earnings, showing all the data for each person. Also calculate the total earnings and print that value at the end of the list of employees.
5. Guess my weight.  
At the county fair a man stands around trying to guess people's weight. You've decided to see how accurate he is, so you collect some data. These data are a set of number pairs, where the first number in the pair is the actual weight of a person and the second number is the weight guessed by the man at the fair. You decide to use two different error measures in your analysis: absolute error and relative error. Absolute error is defined as  $E_{abs} = W_{guess} - W_{real}$ , where  $W_{guess}$  and  $W_{real}$  are the guessed and real weights, respectively. The units of this error are pounds. The relative error is defined by  $E_{rel} = 100 \times E_{abs}/W_{real}$ , where the result of this equation is a percentage. Write a program that will input the set of weight pairs you accumulated, using a function with a sentinel loop to read the data. The number of weight pairs should be between 1 and 100. Write another function that will calculate both the absolute and relative errors of the guesses and display them in a table. Finally, compute and print the average of the absolute values of the absolute errors and the average of the absolute values of the relative errors.
6. Having fen.  
Ms. Honeywell, an American businessperson, is preparing for a trip to Beijing, China, and is worried about keeping track of her money. She will take a portable computer with her, and wants a program that will sum the values of the Chinese fen (coins and bills) she has and convert the total to American dollars. Fen come in denominations of 1, 5, 10, 20, 50, 100, 200, 500, 1,000, and 2,000. The exchange rate changes daily and is published (in English) in the newspaper and on television. Write a program for Ms. Honeywell that will prompt her for the exchange rate and then for the number of fen she has of each denomination. Total the values of her fen and print the total as well as the equivalent in American dollars. Implement the table of fen values as a global constant array of integers.
7. A function defined by a table.  
Write a function that computes a tax rate based on earned salary according to the following table. In this function, if the salary value is not given exactly, use the rate for the next lower salary in the table. For example, use the rate 20% for \$38,596.

Salary (\$)	Tax Rate (%)
0	0
10,000	5
20,000	12
30,000	20
40,000	33
50,000	38
60,000	45
70,000	50

Write a small program that will input a salary from the user, call your function to compute the tax rate, and then print the tax rate and the amount of tax to be paid. Validate the input salary so that it does not fall outside of the salary range in the table.

8. Moving average.

Some quantities, such as the value of a stock, the size of a population, or the outdoor temperature, have frequent small fluctuations but tend to follow longer-term trends. It is helpful to evaluate such quantities in terms of a moving average; that is, the average of the most recent  $N$  measurements. ( $N$  normally is in the range 3 . . . 10.) This technique “smoothes out” the most recent fluctuations, exposing the overall trend. Write a program that will compute a moving average of order  $N$  for the price of a given stock on  $M$  consecutive days, where  $M > N + 4$ . To do this, first read the values of  $N$  and  $M$  from the user. Next read the first  $N$  prices and store them in an array. Then repeat the process below for the remaining  $M - N$  prices:

- (a) Compute and print the average of the  $N$  prices in the array.
- (b) If all  $M$  values have been processed, quit and print a termination message.
- (c) Otherwise, eliminate the value in array slot 0 and shift the other  $N - 1$  values one slot leftward.
- (d) Read a new value into the empty slot at the end of the array.

9. A bidirectional sort.

Another sorting algorithm is similar to the selection sort, the “cocktail shaker” sort. This algorithm differs from the selection sort in the way it selects the next item from the array. Our selection sort always picks the maximum value from the remaining values and swaps it into the beginning of the sorted portion. For the cocktail shaker sort, the first pass finds the maximum data value and moves it to one end of the array. The second pass finds the minimum remaining value and moves it to the other end of the array. Subsequent passes alternate choosing the maximum and minimum values from the remaining data and moving that value to the appropriate end of the array. Eventually the two ends meet in the middle and the data are sorted. Write a program that implements the cocktail shaker sort just described and uses it to sort data sets containing up to 100 values.

# Chapter 11

## An Introduction to Pointers

In this chapter, we introduce the final remaining primitive data type, the pointer, which is the address of a data object. We show how to use pointer literals and variables, explain how they are represented in the computer, and present the three pointer operators: `&`, `*`, and `=`. We explain how, using pointer parameters, more than one result can be returned from a function. Pointers are covered here only at an introductory level and will be considered in greater depth in later chapters.

### 11.1 A First Look at Pointers

A **pointer** is like a pronoun in English; it can refer to one object now and a different object later. Pointers are used in C programs for a variety of purposes:

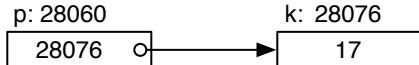
- To return more than one value from a function (using call by value/address).
- To create and process strings.
- To manipulate the contents of arrays and structures.
- To construct data structures whose size can grow or shrink dynamically.

In this chapter we study only the first of these uses of pointers; the others will be explored in Chapters 12, 13, 16, and 16.

#### 11.1.1 Pointer Values Are Addresses

A pointer value (also called a *reference*) is the address (i.e., a specific memory location) of an object. A pointer variable can store different references at different times. If `p` is a pointer variable and the address of `k` is stored in `p`, then we say `p points at k` or `p contains is a reference to k` or `p refers indirectly to k`. In the other direction, we say that `k is pointed at by p` or `k is the referent of p`. In object diagrams, we represent pointer values as arrows and pointer variables as boxes from which arrows originate. The tail of each pointer arrow is a small circle that can be “stored” in a pointer variable; the head of the arrow points at its referent. Figure 11.1 shows a pointer variable named `p` that points at the integer variable `k`, which itself has a value of 17. Figure 11.1 shows a simplified way to diagram pointer variables, without the explicit memory addresses.

**Pointer variables.** To declare a **pointer variable**, we start with the **base type of the pointer**; that is, the type of object it can reference. After the type comes a list of pointer variable names, each preceded by an asterisk, as shown in Figure 11.3. A common mistake is to omit the asterisk in front of `p2`. This makes it a simple integer rather than a pointer. The asterisk must be written before each name, not just appended to the base type. A pointer can refer only to objects of its base type. Although a given pointer can refer to different objects at different times, we cannot use it to refer to an `int` at one moment and a `double` later. Therefore, both `p1` and `p2` in the diagram can be used to refer to an integer variable `k`, but neither could refer to a `char` or a `double`. When we use pointers in expressions, the base type of the pointer lets C know the actual type of the values that can be referenced, so that it can compile appropriate operations and conversions.



p is a pointer variable stored at address 28060.  
 k is an integer variable stored at address 28076.  
 The contents of k is the integer 17.  
 The contents of p is the address of k, which is 28076.  
 The arrow is a reference to k  
 We say that p points at k or p refers to k  
 We say that k is the referent of p.

Figure 11.1. A pointer and its referent.

**Pointer initialization.** When a pointer is declared without an initializer (as an **uninitialized pointer**), memory is allocated for it but no address is stored there, so any value previously stored in that memory location remains. Therefore, a pointer always points at *something*, even when that thing is not meaningful to the current program. It could be an actual object in the program, a random memory address in the middle of the code, or an illegal address that does not even correspond to a memory location the program is allowed to access. Most C compilers will not detect or give error comments about uninitialized pointers. If a program unintentionally uses one, the consequences will not be discovered until run time and can be quite unpredictable, depending on the random contents of memory when the program begins execution. Anything can happen, from apparently correct operation to strange output results to an immediate program crash. To emphasize the unknown consequences of using such pointers, we diagram an uninitialized pointer as a wavy arrow that ends at a question mark, as in the middle diagram in Figure 11.2.

**The NULL pointer.** Many data types include a literal value that means “nothing”: for type `double` this value is 0.0, for `int` it is 0, and for `char` it is \0. There also is a “zero” value for pointer types, the **NULL pointer**, and it is defined in `stdio.h`. We store the value `NULL` in a pointer variable as a sign that it points to nothing. `NULL` is represented in the computer as a series of 0 bits and, technically, is a pointer to memory location 0 (which contains part of the operating system). In diagrams, we represent `NULL` using the electric “ground” symbol, as shown in the rightmost part of Figure 11.2, or an arrow that loops around, crosses itself, and ends in midair. One of the basic uses of `NULL` is to initialize a pointer, to avoid pointing at random memory locations. We often initialize pointers to `NULL` (see Figure 11.4). This indicates that the pointer refers to nothing, as opposed to something undefined.

We diagram a pointer variable as a box containing an arrow (a pointer). Here, we diagram three pointer variables. The first points at an integer, the second is uninitialized, and the third contains the `NULL` pointer.

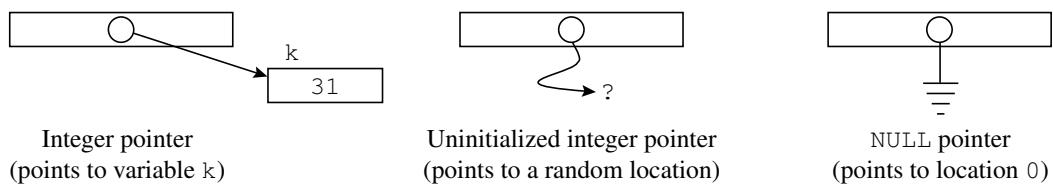


Figure 11.2. Pointer diagrams.

Two uninitialized pointer variables are declared.

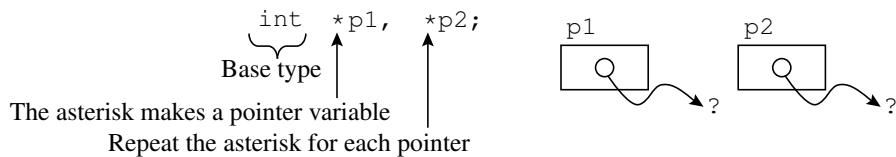
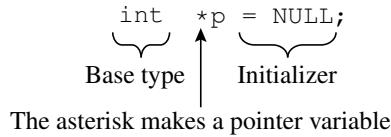


Figure 11.3. Declaring a pointer variable.

---

An integer pointer variable is declared and initialized to NULL.




---

**Figure 11.4. Initializing a pointer variable.**

**Which nothing is correct?** A pertinent question is, What is the difference between 0.0, 0, \0, and NULL? First, even though all are composed entirely of 0 bits, they are of different lengths. A `double` 0.0 often is 8 bytes long, but the character \0 is only 1 byte long. The NULL pointer is the length of pointers on the local system, which, in turn, is determined by the number of bytes required to store a memory address on that system. Second, these zero values have different types and a C compiler treats them like other values of their type when it produces compile-time error comments. For example, if the value 3.1 would be legal in some context, then the value 0.0 also would be legal. The value 0 would be acceptable and require conversion, while the value NULL would be inappropriate and cause a compile-time error.

**The implementation of pointers.** A pointer variable is a storage location in which a pointer can be stored. The pointer itself is the address of another variable. Thus, a *pointer variable* has an address and also contains an address; a *pointer* is the address of its referent. The dual nature of pointers can be confusing, even to experienced programmers. Sometimes it helps to understand how pointers actually are implemented in the computer at run time. The basics are illustrated in Figure 11.5. There, we declare two integer variables and two integer pointers, then diagram the variables created by the declarations. (We assume that an `int` fills 2 bytes and a pointer 4.) Hypothetical memory addresses are shown above the boxes to help explain the actions of certain pointer operations in the next section.

### 11.1.2 Pointer Operations

C has three basic operators<sup>1</sup> that deal with pointers: `&`, `*`, and `=`. In addition, `scanf()` supports a format specifier for printing pointer values. While each operation is straightforward, sometimes the use of pointers can be confusing.

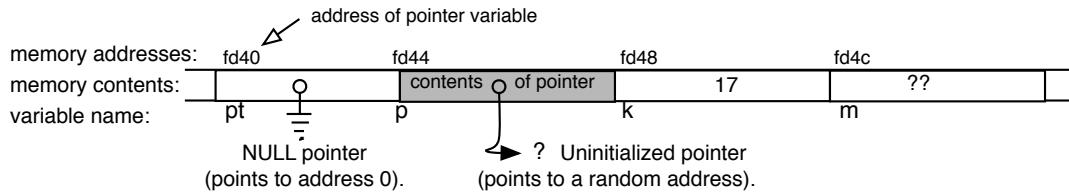
---

<sup>1</sup>Pointers to aggregate types follow different syntactic rules than pointers to simple variables and use an additional operator. The differences are explained in Chapters 13 and 16.

---

```
int *pt = NULL;           // An int pointer variable, initialized to NULL.
int *p;                  // A pointer variable that can point at any int.
int k = 17;               // An integer variable initialized to 17.
int m;                  // An uninitialized integer variable.
```

Storage for these variables might be laid out in the computer's memory as follows:




---

**Figure 11.5. Pointers in memory.**

**References and indirection.** The expression `&k` (the & of a variable) gives us the address of the variable `k`, also called a *reference to k*. The **dereference operator**, `*`, also called **indirection**, is the inverse of `&`; that is, `*p` means the referent of `p` (the object at which `p` points). Since, by definition, `&` and `*` are inverse operations, `*&k == k` and `&*p == p`.<sup>2</sup>

The expression `*p` stands for the referent the same way a pronoun stands for a noun. If `k` is the referent of the pointer `p`, then `m = *p` means the same thing as `m = k`. We say that we *dereference p to get k*. Similarly, `*p = n` means the same thing as `k = n`. Longer expressions also can be written; anywhere that you might write a variable name, you can write an asterisk and the name of a pointer that refers to the variable. For example, `m = *p + 2` adds 2 to the value of `k` (the referent of `p`) and stores the result in `m`. Since the dereference operator and the multiply operator use the same symbol, `*`, the C compiler distinguishes between them by context. If the operator has operands on both the left and right, it means “multiply.” If it has only one operand, on the right, it means “dereference.”

**Pointer assignment.** As just seen, a pointer can be involved in an assignment operation in three ways:

1. We can make an assignment directly *to* a pointer, as in `p = &k`.
2. We can access a value *through* a pointer and assign it to a variable, as in `m = *p`.
3. We can use a pointer to make an *indirect assignment* to its referent, as in `*p = m+2`.

Direct assignments are useful for string manipulation (Section 12.1). Indirect assignment and access through a pointer are used with call by value/address parameters (Section 11.2). Figure 11.6 illustrates the three kinds of pointer assignment using the variables declared in Figure 11.5.

**Notes on Figure 11.6. Pointer operations.** To show the actual relationship between a pointer variable, its contents, and its referent, the addresses and contents of each pointer and variable in this program have been printed and diagrammed.

**First box: pointer declarations.** Figure 11.5 contains a diagram of the initial contents of the two pointers and the variables declared here.

**Second box: direct pointer assignments.** There are two ways to assign a value to a pointer: assign either the address of a variable of the correct base type or the contents of another pointer of a matching type.

- The base type of `p` is the same as the type of `k`, so we are permitted to make `p` refer to `k` with the assignment `p = &k`. In the diagram below the output, note that the address of `k` is written in the variable `p` and that the arrow coming from `p` ends at `k`. We say that `p` *refers to* (or *points at*) `k`.
- Pointers `p` and `pt` are the same type, so we can copy the contents of `p` into `pt` with the assignment `pt = p`. This causes the contents of `p` (which is the address of `k`) to be copied into `pt`. In the diagram, you can see that both `p` and `pt` contain arrows with heads pointing at `k`.

**Third box: indirect pointer assignments.**

- The first line dereferences `p` to get `k`, then fetches the value of `k` and adds 2 to it. The result (19) is stored in `m`; the value stored in `k` is not changed at this time. It is not necessary to use parentheses in this expression because `*` has higher precedence than `+`.
- The second line copies the value of `m` into the referent of `p`, which still is `k`. This changes the value of `k` from 17 to 19, as diagrammed.

**Fourth box: Output.**

The output shows how memory is laid out in our computer, which is running Gnu C<sup>3</sup> and uses 4-byte integers. The memory addresses are printed using the `%p` format specifier, which prints numbers in unsigned hexadecimal notation, with a leading `0x`. Compare this diagram and the output to the memory diagram in Figure 11.5.

<sup>2</sup>The combinations `*&k` and `&*p`, therefore, are silly and not written in a program.

<sup>3</sup>This is the open-code C compiler from the Free Software Foundation.

## 11.2 Call by Value/Address

Most parameter values are passed from the caller to a function using **call by value**; that is, by copying the argument value from the caller's memory area into the parameter variable within the function's memory area. However, there are three situations in C in which copying the argument value is not done, is inefficient, or cannot do the job that is required:

1. When an array of any size is being passed (discussed in Chapter 10).
2. When a function needs to return more than one result (discussed in this section).
3. When a large structure is being passed (discussed in Chapter 13).

All these situations are handled in C by passing an address, not a value, to the function.

This short program connects the program fragments from this section and adds output statements that show the contents and addresses of the variables.

```
#include <stdio.h>

int main( void )
{
    int * pt = NULL;      // An int pointer variable, initialized to NULL.
    int * p;              // A pointer variable that can point at any int.
    int k = 17;            // An integer variable initialized to 17.
    int m;                // An uninitialized integer variable.

    p = &k;               // Use & to set a pointer to k's memory location.
    pt = p;               // Copy a pointer.

    m = *p + 2;           // Add 2 to the value of p's referent; store result in m.
    p = m;               // Copy the value of m into p's referent.

    printf( "address of p: %p contents of p: %p\n", &p, p );
    printf( "address of pt: %p contents of pt: %p\n", &pt, pt );
    printf( "address of k: %p contents of k: %i\n", &k, k );
    printf( "address of m: %p contents of m: %i\n", &m, m );

    return 0;
}
```

When compiled and run on a system with 4-byte integers, the following output was produced:

```
address of p: 0xbffffd44  contents of p: 0xbffffd48
address of pt: 0xbffffd40  contents of pt: 0xbffffd48
address of k: 0xbffffd48  contents of k: 19
address of m: 0xbffffd4c  contents of m: 19
```

This corresponds to the memory layout that follows. Here, the memory addresses are shown in hexadecimal notation and only the last four digits are shown.

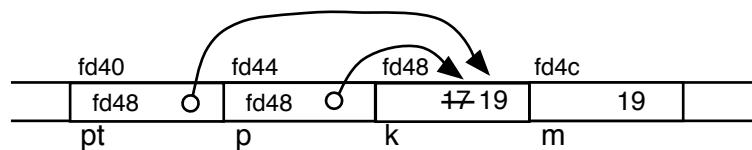


Figure 11.6. Pointer operations.

Chapter 10 discusses call by reference, which is used to pass array arguments<sup>4</sup>. In this method, we pass only the address of the first array slot, not the array's list of values, into the function. Within the function, the reference is transparent to the programmer. That is, references to the array, with or without subscripts, are written exactly the same way in the function as they are in the caller. This makes a large amount of data available to the function efficiently and allows the function to store information into the array. As demonstrated in Section 10.4, a program can pass an empty array into a function, which then fills it with information. When the function returns, that information is in the array and can be used by the caller.

This chapter discusses **call by value/address**. This is a special case of call by value in which the argument is the address of a variable or a pointer to a variable in the caller's memory area. This argument must be stored in a pointer parameter in the function's area. This gives the function full access to the variable that belongs to the caller. This much is exactly like call by reference. However, the similarity ends there because call by value/address is *not* transparent to the programmer; an *indirection* or *dereference* operator must be used in the function's code to access the underlying argument in the storage area of the caller. The remainder of this section explains, in detail, how this works and how to use it.

### 11.2.1 Address Arguments

When call by value is used to pass an address argument, the function receives a reference to the caller's variable and can read from and write to that variable. By reading the caller's variable, the function obtains the value placed there by the calling program. By writing (storing) into the caller's variable, the function can pass a value back to the calling program. When the function ends, the value that it wrote is still in the caller's variable.

You already are familiar with one function that requires an address argument: `scanf()`. When calling `scanf()`, we use an & with arguments of simple types such as `long` and `double`. This technique is not limited to `scanf()`. In any function, we can pass the address of a simple variable by writing an & followed by the name of the variable. The function must have a corresponding parameter of a pointer type.

Another way to pass the address of a simple variable is to write the name (with no ampersand) of a pointer variable that refers to it. In this case, the value of the pointer, which is the address of its referent, is passed.

### 11.2.2 Pointer Parameters

A function declares that it is expecting an address argument from the caller by declaring the corresponding parameter with a pointer type.

When the argument is an array, as in the statistics program of Figures 11.12 through ??, the corresponding parameter can be declared either as a pointer or as an array with an unspecified dimension. For example, if the actual argument were an array of integers, a formal parameter named `ara` could be declared as either `int* ara` or `int ara[]`. The two declarations are identical in most respects, but it is cleaner style to use the latter for arrays. In either case, the parameter name can be used with subscripts within the function to refer to the array elements.

For nonarrays, a **pointer parameter** is declared with an asterisk. When the function is called, the address argument is copied into the corresponding pointer parameter. The base type of the pointer parameter must match the type of the address argument. For example, if a function expects to receive the address of an integer variable, it should declare the corresponding parameter to be of type `int*` (as in the function `f1()` in Figure 11.7). The result is an in-out parameter.

Sometimes an address argument is used to pass a large data structure efficiently<sup>5</sup>. In this case, we may want to have an input parameter that does not permit outward flow of information. To achieve this, the parameter is declared as a constant pointer. For example, `const int * xp` (as in the function `f3()` in Figure 11.7).

**Notes on Figure 11.7. Call by value/address.** Here we have two simple functions that illustrate two techniques for altering the value of a variable in the caller's memory area.

**First box: prototypes.** Functions `f1()` and `f2()` have the same prototype, having one parameter that is an integer pointer. The pointer permits the function to export information. C does not provide a way to restrict a parameter to output-only. In function `f2()`, the usage is output-only, but the syntax would permit two-way flow of information.

---

<sup>4</sup>Call-by-reference is much more widely used in Java and in C++.

<sup>5</sup>This will be discussed in Chapter 13.

---

This program illustrates syntax for call by value/address and gives examples of in, in-out and output parameters.

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

void f1( int * xp );           // uses an in/out parameter
void f2( int * xp );           // uses an output parameter
double f3( const int * xp );   // uses an in parameter

int main( void )
{
    int k = 1;
    double answer;

    puts( "\n Call by value/address Demo." );

    printf( "Original value of k: %i\n", k );
    f1( &k );                  // This function changes the value of k.
    printf( "After f1(), changed value of k: %i\n", k );

    printf( "f2( &k );          // This function changes the value of k.\n" );
    printf( "After f2(), input is stored in k: %i\n", k );

    answer = f3( &k );         // This function cannot change k.
    printf( "After f3(), the square root of %i = %.3f\n", k, answer );

    return 0;
} // -----
void f1( int * xp )           // xp is an in/out parameter
{
    *xp = *xp + 2;            // add 2 to the old value of xp's referent.
}

// -----
void f2( int * xp )           // xp is an output parameter.
{
    printf( "Enter an integer: " );
    scanf( "%i", xp );
}

// -----
double f3( const int * xp )   // xp is an in parameter.
{
    return sqrt( *xp );       // use xp's referent (no change).
}
```

---

Figure 11.7. Call by value/address.

---

This version of swap does not work because call by value is used to pass the parameters.

```
#include <stdio.h>
void badswap( double f1, double f2 );
int main( void ) {
    double x = 10.2, y = 7;
    printf( "Before badswap: x=%5.1g y=%5.1g\n", x, y );
    badswap( x, y );
    printf( "After badswap: x=%5.1g y=%5.1g\n", x, y );
}

void badswap( double f1, double f2 ) {
    double swapper = f1;
    f1 = f2;
    f2 = swapper;
}
```

---

**Figure 11.8. A swap function with an error.**

***Second box and function f1: indirect reference through an in-out parameter.***

- We pass `&k`, the address of `k`, as the argument to the pointer parameter in `f1()`. This address will be stored in the memory location for `xp` when the call occurs, so `xp` will refer to `k`.
- We use the initial value of `k` in this function by writing `*xp`. After adding 2, we store the result back into `k` by writing `*xp =`
- We call `xp` an *input* parameter because we use the information that the caller stored in it. We call it an *output* parameter because we change that information. Thus, it is an in-out parameter.
- The value of `k` is displayed before and after the function call to show that the call both used and changed the value of `main`'s variable. (See the output, below.)

***Third box and function f2: using an output parameter.***

- We use `xp`, an output parameter here, to return a value from `scanf()` back to the caller. The new value is printed in `main()` after the function call.
- We want to pass the address of `k` to `scanf()` so that input can be stored in `k`. We could do this by writing `&xp` as the argument, but this simplifies to just `xp`, which contains the original address of `k` as it was passed into the function. The `scanf()` call stores a value directly into `main()`'s variable `k`.

***Fourth box and function f3: using a const \* input parameter.***

- Sometimes our data is stored in large structures (presented in Chapter 13) that occupy many bytes of memory. It is undesirable to copy the whole structure because of the time and the space that would consume. In such a situation we use call by value/address. However, we also want to protect the caller's variable against the possibility of being changed by the function or by any other function it might call. To do this, we can use a `const pointer` parameter, also known as a *read-only parameter*.
- In this function, we use a `const *` parameter. This lets us use but not change the value of `main`'s variable. To access that value, we write `*xp`.

***Sample output.***

```
Original value of k: 1
After f1(), changed value of k: 3
Enter an integer: 7
After f2(), input is stored in k: 7
After f3(), the square root of 7 = 2.646
```

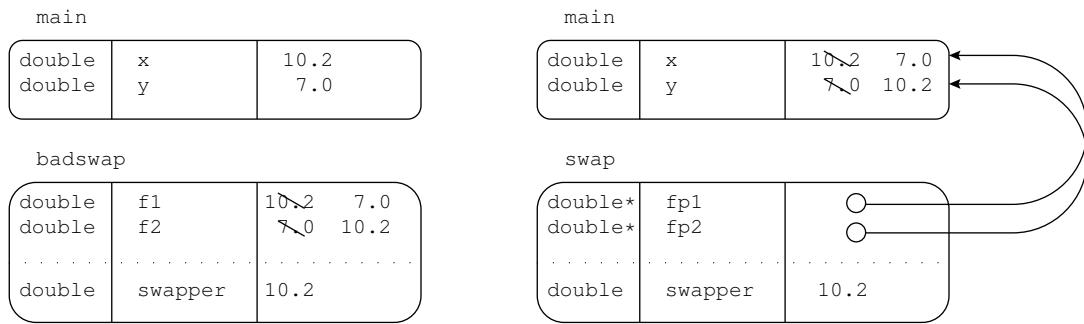


Figure 11.9. Seeing the difference.

### 11.2.3 A More Complex Example

In some situations, call by value does not provide enough information to enable a function to do its task. The simplest example consists of a function that wants to swap the values of its two parameters. In Figures 11.8 through ??, we examine two possible versions of this simple **swap function**. The first fails to swap the values; the second works properly.

The memory use for Figure 11.8 is diagrammed on the left. Each function has its own memory area, and values of the arguments are copied from variables of `main()` into the parameters of `badswap()`. Assignments change only the values in the parameters.

The memory use for Figure 11.10 is diagrammed on the right. The arguments here are pointers to variables of `main()`. Indirect assignments made through these pointers change the underlying variables.

#### Notes on Figures 11.8, 11.10, and 11.9. Seeing the difference in the swap functions.

**First and second boxes, Figure 11.8: the first version of the swap.** The first box declares two parameters as type `double`, not `double*`, so the arguments will be passed by value. When `main()` calls `badswap(x,y)` (second box), the current values of `x` and `y` are copied into `badswap()`'s parameters `f1` and `f2`. The function receives these values, not the addresses of the variables. This is shown in the diagram for `badswap()` on the left side of Figure 11.9.

**Third box, Figure 11.8: the bad swap.** When `badswap()` swaps the values, it swaps the copies stored in the parameters, not the originals. In the diagram, note that the assignments in `badswap()` cause no changes to the variables of `main()`. The program output is

```
Before badswap: x= 10.2  y= 7.0
After badswap: x= 10.2  y= 7.0
```

This version of swap works because call by value/address is used to pass the parameters.

```
#include <stdio.h>
void swap( double * fp1, double * fp2 );
int main( void ) {
    double x = 10.2, y = 7;
    printf( "Before swap: x=%5.1g  y=%5.1g\n", x, y );
    swap( &x, &y );
    printf( "After swap:  x=%5.1g  y=%5.1g\n", x, y );
}
void swap( double * fp1, double * fp2 ) {
    double swapper = *fp1;
    *fp1 = *fp2;
    *fp2 = swapper;
}
```

Figure 11.10. A swap function that works.

**Problem scope:** Calculate the arithmetic mean, variance, and standard deviation of  $N$  experimentally determined data values.

**Input:** The user will specify the number of data values,  $N$ , then  $N$  data values will be typed in. These will be real numbers.

**Restrictions:** No more than 50 data values will be processed.

**Formulas:**

$$\begin{aligned} \text{Mean} &= \frac{\sum_{k=1}^N x_k}{N} \\ \text{Variance} &= \frac{\sum_{k=1}^N (x_k - \text{mean})^2}{N-1} \quad \text{for } N < 20 \\ \text{Variance} &= \frac{\sum_{k=1}^N (x_k - \text{mean})^2}{N} \quad \text{for } N \geq 20 \\ \text{Standard deviation} &= \sqrt{\text{Variance}} \end{aligned}$$

**Output required:** The mean, variance, and standard deviation of the  $N$  points, accurate to at least two decimal places.

Figure 11.11. Problem specifications: Statistics.

**First and second boxes, Figure 11.10: the good version of swap.** In contrast, this version uses call by value/address (first box). The arguments are two addresses (second box), which are stored in the `swap()` parameters `fp1` and `fp2`. In the diagram, you can see that the parameters of `swap()` are pointers to the variables of `main()`.

**Third box, Figure 11.10: the good swap.** The function receives pointers to the variables, not copies of their values. When `swap()` says `swapper = *fp1`, it copies the value of the referent of `fp1` into `swapper`. When it executes `*fp1 = *fp2;`, it copies the value of the referent of `fp2` into the referent of `fp1`. This changes the value of the corresponding variable in `main()`, not the pointer in the `swap()` parameter. We say that `*fp1 = *fp2;` fetches the value *indirectly through fp2* and stores it *indirectly through fp1*. The final output from this program is

```
Before swap: x= 10.2  y=  7.0
After swap:  x=  7.0  y= 10.2
```

### 11.3 Application: Statistical Measures

In many experiments, the measured values of the experimental variable are distributed about the mean value in a bell-shaped curve centered on the **mean value of the array**; that is, the arithmetic average of the data values. Such a distribution is called a *normal, or Gaussian, distribution*.

The **variance** and **standard deviation** of a set of data are measures of how significantly the measured values differ from the true mean of the distribution. To compute these measures accurately, we need at least 20 data values. For fewer than 20 data points, we use the slightly different formulas, shown in Figure 11.11, to estimate the variance and standard deviation. In these equations,  $x_1, x_2, x_3, \dots, x_N$  are the data values and  $N - 1$  is called the *degree of freedom* of the data.

In the next program example, we introduce a method for computing these statistical measures for  $N$  data values:  $x_1, x_2, x_3, \dots, x_N$ .<sup>6</sup> The program specification is given in Figure 11.11 and the main program is shown in Figure 11.12. To keep the flow of logic in all parts of the program simple and uniform, major phases of the computation have been written as separate functions that work with a data array. We call `get_data()` to read the data into an array, then pass that array to the `stats()` function, and finally, print the answers. A call graph is given in Figure 11.13 and the two array-processing functions are found in Figure 11.14.

<sup>6</sup>J. P. Holman, *Experimental Methods for Engineers*, 7th ed. (New York: McGraw-Hill, 2001).

The specification for this program is in Figure 11.11 and the call graph is in Figure 11.13. The three programmer-defined functions are in Figures ?? and ??.

```
#include <stdio.h>
#include <math.h>                                // For sqrt()

#define N    50                                     // Maximum number of data values.

void get_data( double x[], int n );
void stats( const double x[], int n, double * meanp, double * variancep );

int main( void )
{
    double x[N];                                // An array for the N data values.
    int num;                                     // Actual number of data values.

    double mean;                                 // The mean of the values in array x.
    double var;                                  // Variance of the data in array x.

    double stdev;                               // Standard deviation of the data in array x.

    puts( "\n Computing statistics on a set of numbers." );
    printf( "\n Enter number of values in data set (2..%i): ", N );
    for (;;) {
        scanf( "%i", &num );
        if (num > 1 && num <= N) break;
        printf( "Error: %i is out of legal range, try again: \n", num );
    }
    printf( " Computing statistics on %i data values.\n", num );

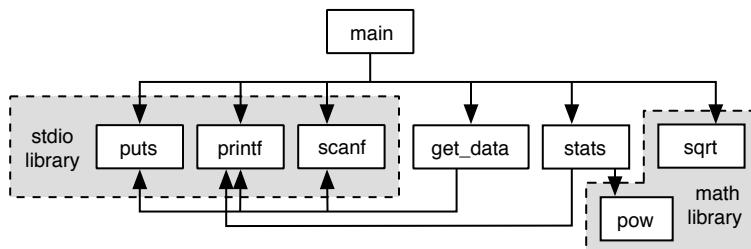
    get_data ( x, num );
    stats( x, num, &mean, &var );

    stdev = sqrt(var);

    printf( "\n\n The mean of the %i data values is = %.2f \n", num, mean );
    printf( " The variance is = %.2f\n", var );
    printf( " The standard deviation is = %.2f \n", stdev );
    return 0;
}
```

**Figure 11.12. Mean and standard deviation.**

This graph is for the program found in Figures 11.12 and 11.14.



**Figure 11.13. Call graph for the mean and standard deviation program.**

Notes on Figure 11.12. Mean and standard deviation.

*First and third boxes: the data array and the definition of  $N$ .*

Every part of this program uses the value of  $N$  to define the number of data values that are to be read, processed, or output. We easily can increase or decrease the length by changing only the `#define` at the top of the program. This is one of the most important ideas in computing: By using loops and arrays, we can process a virtually unlimited number of data items. Files that store large amounts of data are the final element needed in this application. We will revisit this example in Chapter 14 to show how such data can be read from a file.

It is rare that the maximum number of data values is used, since the value of  $N$  usually is set to a comfortably large value. Therefore, the user needs to specify the size of the current data set. This value, rather than  $N$ , will serve as the processing limit in each of the array functions.

*Second box: the function prototypes.*

These functions are called from the main program in Figure 11.12. After reading a set of  $n$  experimentally determined data values into an array,  $x$ , we use summing loops to calculate the mean and variance of those values.

```
// -----
// Given an empty array of length n, input data values to fill it. */
void get_data( double x[], int n )
{
    int k;                      // Loop counter and subscript.
    puts( "Please enter data values when prompted." );
    for (k = 0; k < n; ++k) {
        printf( "x[%i] = ", k ); // Prompt for kth value.
        scanf( "%lg", &x[k] );   // Read into array slot k.
    }
}

// -----
void stats( const double x[], int n, double * meanp, double * variancep )
{
    int k;                      // Counter and array subscript for both loops.
    double sum;                  // Accumulator for both loops.
    double divisor;              // From the definition of variance.
    double local_mean;           // Local copy of first answer.

    for (sum = k = 0; k < n; ++k) {
        printf( "\n      x[%d] = %.2f", k, x[k] );
        sum += x[k];
    }
    *mean = local_mean = sum / n; // Store average locally, also return it.

    for (sum = k = 0; k < n; ++k) {
        sum += pow( (x[k] - local_mean), 2 );
    }
    if (n < 20) divisor = n-1;
    else divisor = n;
    *variance = sum / divisor;   // Return the variance.
}
```

Figure 11.14. The `get_data()` and `stats()` functions.

These are the prototypes for the functions in Figure 11.14. Both of these functions have an array parameter and an integer parameter that gives the size of the data set. In `stats()`, we restrict the array to input-only by writing `const` as part of the parameter type. We do this because `stats()` needs to use, but not to modify, the array values. In addition, `stats()` has two output parameters. Note that the square brackets must be used for an array in the parameter declaration, but they are omitted in the function call.

#### *Fifth box: the function calls.*

- The function calls must correspond to the prototypes. Each call must have the same number of arguments (of the same type and in the same order) as the parameters declared by the prototype. The pairs are
  - In `get_data()`, `x` is an array parameter which is passed by reference into a `double[]` parameter. Because `x` is an array, this is a reference parameter, and therefore is in/out. However, it will be used in an output-only style, to carry the data back to `main()`.
  - In `stats()`, `x` is an array parameter which is passed by reference into a `const double[]` parameter. This is an input parameter. Because it is a reference parameter, the function will be able to read input from the array. Because of the `const`, the function will not be able to change the data in the array.
  - `num`, in both functions, the length of the array, which is passed by value into an `int` parameter. This is an input parameter; call by value with an argument of a simple type does not support outward flow of information.
  - `meanp` and `variancep` in `stats()` are used as output parameters. In the call, we write `& mean` and `& variance` to pass references into function, where they are stored in the parameters. The parameter `meanp` is type `double*`, which is appropriate for storing the address of a `double` variable.

#### *Sixth box: Using the returned value.*

After returning from `stats`, the variables `mean` and `var` contain meaningful values. We now call `sqrt(var)` to compute the standard deviation, then print the statistics.

#### *Output.*

The `main()` function prints headings, reads the number of data items, calls the four functions, and then prints the answers. Following is sample output:

```
Computing statistics on a set of numbers.

Enter number of values in data set (2..50): 5
Computing statistics on 5 data values.
Please enter data values when prompted.
x[0] = 77.89
x[1] = 76.55
x[2] = 76.32
x[3] = 79.43
x[4] = 75.12

      x[0] = 77.89
      x[1] = 76.55
      x[2] = 76.32
      x[3] = 79.43
      x[4] = 75.12

The mean of the 5 data values is = 77.06
The variance is = 2.72
The standard deviation is = 1.65
```

#### **Notes on Figure 11.14. The `get_data()` and `stats()` functions.**

##### *The `get_data()` function.*

The array parameter, `x`, is an output parameter that is passed by reference. When control enters this function, the parameter array is empty. After the last data value has been read, control returns to the caller and the caller can use the values stored in the array by the function.

##### *First box: variables for the `stats()` function.*

The third parameter will be used to send one answer (the average) back to the caller. But we also define a local variable for the average to make it easier and more efficient to use in later calculations.

***Second box: the average.***

This is the usual loop for computing the average of an array. Once all `n` values have been summed, we can calculate the average and return it to `main()`. The division that computes the mean is after the loop, rather than within it, because there is no need to perform a division on every repetition. We need not worry about division by 0 because `n` has been validated in `main()`.

The average is first stored in a local variable, then returned to the caller in the same statement by writing `*meanp = local_mean`. We keep a local copy of the average because we will use it again, repeatedly, in the next loop.

***Third box: the pow() function.***

The `pow()` function is in the `math` library. It raises a `double` value to a `double` power and returns a `double` result. In this call, the integer 2 will be coerced to type `double` before it is passed to the function. The result will be the square of the first argument.

***Fourth box: returning the second answer.***

Once all `n` squares have been summed, we set the divisor to `n` or `n-1`, according to the specifications, then perform the division and return the result to `main()` by assigning it to `*variancep`.

Unlike the average, we do not store the variance in a local variable after we compute it because we do not need it again in this function.

### 11.3.1 Summary: Returning Results from a Function

We have now demonstrated three ways to return a result from a function:

1. By using the `return` statement.
2. By using a pointer parameter.
3. By storing the results in an array parameter.

The first method is simple and it supports a total separation between the “territory” of the calling program and the actions of the function. The function need not receive an address from the caller to use `return`. This kind of separation is a highly important debugging tool and should be used wherever possible. Unfortunately, the `return` statement is limited to one result.<sup>7</sup>

The second method can be used for multiple results, as in the previous program, but it involves the use of the address-of operator (`&`) in the function call, the dereference operator (`*`) in the function body, and a pointer declaration in the function’s prototype and header. Using these operators is somewhat awkward and can become confusing. Sometimes the required stars and ampersands are omitted accidentally. More significant, though, is that each pointer parameter supplies the function with an address in the memory area that belongs to its caller. Each such address can be a source of unintended damaging interaction between the two code units. Therefore, we prefer to pass parameters by value wherever possible. Even so, call by value/address is quite important in C and frequently must be used.<sup>8</sup>

Finally, returning large numbers of results of the same type is possible by using an array. It generally is convenient, efficient, and not confusing. However, the array parameter still is an address of storage that belongs to the caller. Therefore, an array parameter (like a pointer parameter) reduces the isolation of one part of the program from the other, potentially making debugging harder.<sup>9</sup>

## 11.4 What You Should Remember

### 11.4.1 Major Concepts

- A pointer is an address. If `p` is a pointer, then the base type of `p` determines how the compiler will interpret the data stored in the referent of `p`. For example, if a `float` pointer is dereferenced, the result is treated like a `float`.

---

<sup>7</sup>This one result however can be a complex object, such as a structure that contains multiple components. Structures will be discussed in Chapter 13.

<sup>8</sup>This is a defect of C. It is corrected in C++, which supports a third form of parameter passing, termed *call by reference*.

<sup>9</sup>Clearly, none of these three mechanisms is an ideal solution to the problem. For this reason, the reference parameter, a fourth method of returning results, was implemented in C++.

- There are two major pointer operators, `&` and `*`, which are the inverses of one another. The address operator, `&`, is used to refer to the memory location of its operand. The dereferencing operator, `*`, uses the address in the pointer operand to either retrieve or store a value at that location.
- Pointer variables, like all others, can have garbage in them if they are not initialized. Sometimes this garbage could be an accessible memory location and other times not. The value `NULL` often is used to initialize pointers. Any attempt to reference this location on a properly protected system will cause an immediate and consistent run-time error that can be tracked down more easily than the intermittent errors caused by using a random address.
- When a parameter is a pointer variable, the corresponding argument must be an array, a pointer, or an address. The called function then can store data at that address and, by doing so, change the value in a variable that belongs to the caller. We call this method of parameter passing *call by value/address*.
- When a parameter is a `const` pointer variable, the corresponding argument must be an array, a pointer, or an address. The called function then can use the data at that address but cannot change it.
- Using pointer parameters, one can return multiple results from a single function. An array parameter is translated as a pointer and lets us pass a large amount of data to or from a function efficiently.

### 11.4.2 Programming Style

- In this text, pointer variable declarations place the `*` next to the base type (as in `float* f`) or let it float (as in `float * f`) between the type and the name. However, it is quite common, for various historical reasons, for programmers to use the style `float *f`, in which the asterisk is attached to the variable name. We prefer the style `float* f` because it clarifies that the data type of the variable is a pointer type.
- To avoid confusion about which variables are pointers and which are not, it is best to declare only one pointer per line. This lets you maintain our preceding style convention and provides space for a comment. If you declare more than one pointer on the same line, be sure to use an `*` for each one.
- Use the correct zero literal. For pointers, use `NULL`. Reserve 0 for use as an integer.
- It is good practice to initialize pointers to `NULL`, which makes pointer usage errors easier to find.
- Do not use the operator combinations `&*` and `*&`. Since the operators cancel each other out, there is no need for either.
- In a function definition, put the parameters that bring information into the function first and the call-by-value/address parameters that carry information out of the function last. Any in-out parameters can be placed in between.
- Minimizing the use of call by value/address increases the separation between caller and subprogram, which is helpful when debugging. However, call by value/address is an important mechanism. Learn to use it wisely.
- Do not use a global variable to pass information into or out of a function. In all other cases, pass the information as a parameter. The one exception to this rule is when the global variable is used with a piece of pre-existing code that you cannot change,
- When using both the return value and pointer parameters to return results from a function, use the return value for a result that *always* is meaningful, such as a status code. Use parameters for results that may or may not be meaningful, depending on circumstances.
- If a function, `f()`, is not called by `main()` and is called by only a single other function, then the prototype for `f()` may be written inside the function that calls it. This properly limits its accessibility to the scope the programmer intended.

### 11.4.3 Sticky Points and Common Errors

- The most common pointer error is an attempt to use a pointer that has not been set to refer to anything. Sometimes such pointers have a `NULL` value; sometimes they contain garbage. In both cases, the attempt to use such a pointer is an error. On some systems, this causes an immediate crash. On others, execution may continue indefinitely before anything unexpected happens. Pointers are like pronouns; until they are initialized to point at specific variables, they must not be used. Be careful with your pointers and check them first when a program that uses pointers malfunctions.

- There can be confusion between the multiply and dereference operators. It should be clear from the context which is being used. If, by accident, an operand is omitted in an expression, the compiler might interpret the `*` as multiplication (when dereference was intended) without generating an error message. Using redundant parentheses can help the compiler interpret your code as you intended, but excess parentheses can clutter the expression, potentially causing other kinds of errors. Some compromise in style is needed.
- Do not reverse the use of `&` and `*`. Using the wrong operator always leads to trouble.
- When using call by value/address, a common oversight is to omit some of the required asterisks in the function or the ampersand in the call. This can produce a variety of compile-time error comments that may warn you about a type mismatch between argument and parameter but not tell you exactly what is wrong. For example, if the omission is in the parameter declaration, the error comment actually will be on the first line in the function where that parameter is used. Sometimes a beginner “corrects” the thing that was not wrong, which leads to different errors, and so on, until the code is a mess. When you have a type mismatch error, think carefully about why the error happened and fix the line that actually is wrong. Sometimes drawing a memory diagram can help clear up the confusion.
- Even if you have no type mismatches between arguments and parameters, you may not have the kind of communication you want. If you want call by value/address, you must declare things properly; otherwise, you get functions like `badswap()` in Figure 11.8. Still other times you might omit the `*` where it is needed inside the function, and the address in a pointer parameter will be changed rather than the contents of the other memory location. The compiler may not complain about this, but it certainly will affect the logic of your program. Also, forgetting the `&` in front of an argument for a pointer parameter may not generate a compiler error, but the value of the parameter during execution will be nonsense and usually cause the program to crash. Some compilers give warnings about errors like this.

#### 11.4.4 Where to Find More Information

- Strings pointers and ragged arrays are covered in Chapter 12.
- Array processing using pointers is explained in Chapter 16.
- Pointers to functions are covered in Chapter 16.
- Pointers to dynamic storage allocation areas are found in Chapter 16.

#### 11.4.5 New and Revisited Vocabulary

These are the most important terms, concepts, and keywords presented in this chapter.

pointer	memory address	call by reference
pointer variable	<code>&amp;</code> of a variable	call by value/address
base type of pointer	<code>*</code> operator	address argument
NULL pointer	indirection	pointer parameter
uninitialized pointer	output parameter	swap function
referent	input parameter	mean value of an array
reference	in-out parameter	variance
dereference	call by value	standard deviation

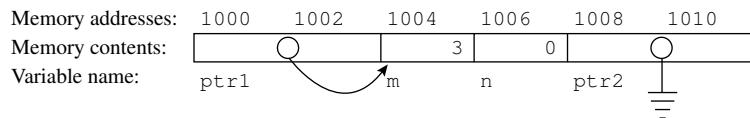
### 11.5 Exercises

#### 11.5.1 Self-Test Exercises

1. Complete each of the following C statements by adding an asterisk, ampersand, or subscript wherever needed to make the statement do the job described by the comment. Use these declarations:

```
float x, y;
float s[4] = {62.3, 65.5, 41.2, 73.0};
float * fp;
```

- (a) `fp = y;` // Make fp refer to y.  
 (b) `fp = s;` // Point fp at slot containing 65.5.  
 (c) `x = fp;` // Copy fp's referent into x.  
 (d) `fp = y;` // Copy y's data into fp's referent.  
 (e) `fp = s[];` // Copy 73.0 into fp's referent.  
 (f) `scanf( "%g", x );` // Read data into x.  
 (g) `scanf( "%g", fp );` // Read data into fp's referent.  
 (h) `printf( "%g", s );` // Print value of second slot.
2. (a) Given the following diagram of four variables, write code that declares and initializes them, as pictured. On this system, an `int` occupies two bytes.



- (b) Now write two or three direct or indirect pointer assignment statements to change the memory values to the configuration shown here:
- |                   |      |      |      |      |      |      |
|-------------------|------|------|------|------|------|------|
| Memory addresses: | 1000 | 1002 | 1004 | 1006 | 1008 | 1010 |
| Memory contents:  |      |      |      |      |      |      |
| Variable name:    | ptr1 | m    | n    | ptr2 |      |      |
3. Consider the function prototype that follows. Write a short function definition that matches the prototype and the description above it. Then write a main program that declares any necessary variables, makes a meaningful call on the function, and prints the answer.

```
// Return true via out1 if in1 == in2, false otherwise.
void same( int in1, int in2, int* out1 );
```

4. All the questions that follow refer to the given partially finished program.

```
#include <stdio.h>
void freeze( int temperatures[], int n );
void show( int temperatures[], int n );

int main( void )
{
    int max = 6;
    int degrees[6] = { 34, 29, 31, 36, 37, 33 };
    freeze( degrees, max );
    printf( "\n After freeze: max = %i\n", max );
    .....
}

// -----
void freeze( int temperatures[], int n )
{
    int k;
    for(k = n-1; k >= 0; --k)
        if (temperatures[k] >= 32) --n;
}
```

- (a) The second parameter of the `freeze()` function is supposed to be a call-by-value/address parameter. However, the programmer forgot to write the necessary ampersands and asterisks in the prototype, call, function header, and function code. Add these characters where needed so that changes made to the parameter `n` actually change the underlying variable `max` in the main program.  
 (b) Write a function named `show()`, according to the prototype given, that will display all the numbers in the array on the computer screen. Write a call on this function on the dotted line in `main()`.  
 (c) Draw a storage diagram similar to the one in Figure 11.9 and use it to trace execution of the call on `freeze()` and the following `printf()` statement. On your diagram, show every value that is changed.

- (d) What is the output from this program after the additions?
5. (Advanced question) Trace the execution of the program that follows. Make a memory diagram following the example of Figure 11.9 and showing each program scope in a separate box. On your diagram, show the value given to each parameter when a function is called, how the values of its variables change as execution proceeds, and the value(s) returned by the function. Show the output produced on a separate part of your page.

```
#include <stdio.h>
int y = 2, z = 3;           // Global variables!

int func1( int* x, int* y );
void func2( int* x ){ *x = y; y = z; z = *x; }

int main( void )
{
    int x = func1( &y, &x );
    printf( "X = %i Y = %i Z = %i\n", x, y, z );
}

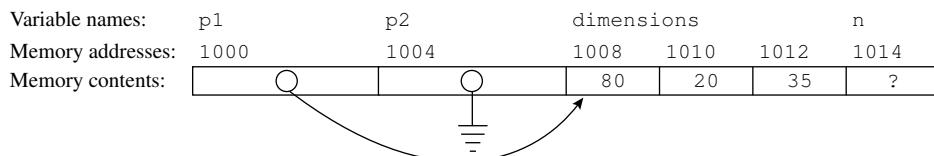
int func1( int* x, int* y )
{
    *y = z+1;
    *x = *y;
    func2( y );
    z = *x+2;
    return *y;
}
```

### 11.5.2 Using Pencil and Paper

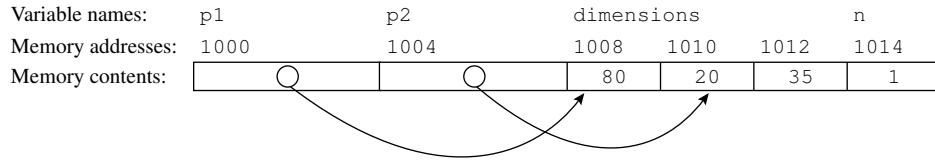
1. Complete each of the following C statements by adding an asterisk, ampersand, or subscript wherever needed to make the statement do the job described by the comment. Use these declarations:

```
short s, t;
short age[] = { 30, 65, 41, 23 };
short *agep, *maxp;
```

- (a) `agep = age;` // Make `agep` refer to first `age` in array  
 (b) `s = agep;` // Copy value of `agep`'s referent into `s`.  
 (c) `agep = age[];` // Copy 65 into `agep`'s referent.  
 (d) `maxp = agep;` // Make `maxp` refer to `agep`'s referent.  
 (e) `agep = (age[]+age[])/2;` // Store mean of 2nd and last ages in `agep`'s referent.  
 (f) `scanf( "%hi", age[] );` // Read into third array slot.  
 (g) `scanf( "%hi", agep );` // Read into `agep`'s referent.  
 (h) `printf( "%hi", agep );` // Print `agep`'s referent.
2. (a) Given the diagram of three variables and an array, write code that declares the variables and initializes the array and the pointers, as pictured.



- (b) Write a function that takes the `dimensions` array as its parameter. Use a nested `if...else` statement to identify the smallest dimension and return the subscript of its slot.  
 (c) Now call the function and store the result in `n`, then use `n` to set `p2` to point at the smallest dimension.



3. Draw a call graph for the bisection program on the text website. Include all programmer-defined and library functions.
4. Consider the function prototype that follows. Write a short function definition that matches the given prototype and description. Write a main program that declares necessary variables, makes a meaningful call on the function, and prints the answer.

```
// Set the referent of dp to its own absolute value.
// Return +1 if it was positive, 0 if it was zero,
// and -1 otherwise.
int signum( double* dp );
```

5. The questions that follow refer to the given partially finished program.

```
#include <stdio.h>
#define MAX 10
void count( int ages[], int adults, int teens );

int main( void )
{
    int family[MAX] = {44, 43, 21, 18, 15, 13, 11, 9, 7};
    int Nadults=0, Nteens=0, Nkids=0;

    puts( "\nFamily Structure" );
    count( family, Nadults, Nteens );
    Nkids = MAX - (Nadults + Nteens);
    printf( "Family has %i adults, %i teens, %i kids\n",
            Nadults, Nteens, Nkids );
    puts( "-----\n" );
}

void count( int ages[], int adults, int teens )
{
    int k;
    for (k = 0; k < MAX; ++k) {
        if ( ages[k] >= 18 ) ++adults;
        else if ( ages[k] >= 13 ) ++teens;
    }
}
```

- (a) The second and third parameters of the `count()` function need to be pointer parameters. However, the programmer forgot to write the necessary ampersands and asterisks in the prototype, call, function header, and function code. Add these characters where needed so that changes made to the parameters `adults` and `teens` actually change the underlying variables in the main program.
- (b) Draw a storage diagram similar to the one in Figure 11.9 and use it to trace execution of the corrected program. On the diagram, show every value changed and show the output on a separate part of the page.
6. Look at the main program and functions for the bisection program on the text website. Fill in the following table, listing the symbols *defined* (not used) in each program scope. List global symbols on the first line. Allow one line below that for each function in the program (`main()` has been started for you).

Scope	Input Parameters	Output Parameters	Variables	Constants
global	—	—		
main()	—	—		
...				

### 11.5.3 Using the Computer

#### 1. More stats.

Start with the statistics program in Figures 11.12 through 11.14. Add a parallel array for the student ID numbers, which should be read as input. Add functions to do these three tasks:

- (a) Find the maximum score and return the score and its array index through pointer parameters.
- (b) Find the minimum score and return the score and its array index through pointer parameters.
- (c) Find the score that is closest to the average and return the score and its array index through pointer parameters.

In `main()`, call your three functions and print the student ID number and score for the best, closest to average, and weakest student.

2. Class average and more.

An instructor has a set of exam scores stored in a data file. Not only does he want a report containing the average and standard deviation for the exam, he wants lots of other statistics. These include the high score, the low score, the median score, and the coefficient of variation, *cv*. The *median score* is defined to be the middle one in the array of scores, if that array is sorted. This *coefficient of variation* relates the “error” measured by the standard deviation to the “actual” value measured by the arithmetic mean as  $cv = \text{stdev}/\text{mean}$ . Write a program that will read, at most, 100 exam scores from a user-specified file and print out the indicated statistics. Use portions of the statistics programs in this chapter, as appropriate.

3. Pointer and referent.

Write a program that creates the integer array `int ara[] = {11, 13, 17, 19, 23, 29, 31}`. Also create an integer pointer `pt` and make it point at the beginning of the array. Write `printf()` statements that will print the address and contents of both `ara[0]` and `pt`. Use this format:

```
printf( "address of pt: \t %p    contents:\t %p\n", &pt, pt );
```

Then write 10 similar `printf()` statements following the same format to print the address and contents of the slots designated by the following expressions: `(*pt+3), *pt, (pt[3]), *&pt, *pt[3], &*pt, *(pt+3), (*pt++), *(pt++), (*pt)++`.

Some of these will cause compile-time errors when printing the address field, the contents field, or both; in such cases, delete the illegal expression and print dashes instead of its value. When the program finally compiles and runs, use the output to complete the following table, grouping together items that have the same memory address:

	Address	Contents
<code>pt</code>		
<code>ara</code>		
...		

Finally, make four lists: (a) illegal pointer expressions, (b) expressions that have identical meanings, (c) expressions that change pointer values, and (d) expressions that change integer values. It will require careful reasoning to get the last two lists correct.

4. Exam grades.

Start with the program in Figures 11.12 through 11.14; modify it as follows:

- (a) In the main program, declare an array to store exam scores for a class of 15 students. Print out appropriate headings and instructions for the user. Call the appropriate functions to read in the exam scores and calculate their mean and standard deviation. Print the mean and standard deviation. Then call the `grades()` function described here to assign grades to the students’ scores and print them.
- (b) Modify the `average()` function so that it does not print the individual exam scores during its processing.
- (c) Write a new function, named `grades()`, with three parameters: the array of student scores, the mean, and the standard deviation. This function will go through the array of exam scores again and assign a letter grade to each student according to the following criteria. Using one line of output per student, print the array subscript, the score, and the grade in columns. The grading criteria are
  - i. A, if the score is greater than or equal to the mean plus the standard deviation.
  - ii. B, if the score is between the mean and the mean plus the standard deviation.
  - iii. C, if the score is between the mean and the mean minus the standard deviation.
  - iv. D, if the score is between the mean minus the standard deviation and the mean minus twice the standard deviation.
  - v. F, if the score is less than the mean minus twice the standard deviation.

If a score is exactly equal to one of these boundary limits, give the student the higher grade.

5. Positive and negative.

Write a function, named `sums()`, that has two input parameters; an array, `a`, of `floats`; and an integer, `n`, which is the number of values stored in the array. Compute the sum of the positive values in the array and the sum of the negative values. Also count the number of values in each category. Return these four answers through output parameters. Write a main program that reads no more than 10 real numbers and stores them in an array. Stop reading numbers when a 0 is entered. Call the `sums()` function and print the answers it returns. Also compute and print the average values of the positive and negative sets.

6. Sorting.

Write a `void` function, named `order()`, that has three integer parameters: `a`, `b`, and `c`. Compare the parameter values and arrange them in numerical order so that  $a < b < c$ . Use call by value/address so the calling program receives the values back in order. In addition, the function `order()` should start by printing the addresses and contents of its parameters, as well as the contents of the locations to which they point. Write a main program that enters three integers, prints their values and addresses, orders them by calling the function `order()`, and prints their values again after the call. Add a query loop to allow testing several sets of integers.

7. Compound interest.

- (a) Write a function to compute and return the amount of money,  $A$ , that you will have in  $n$  years if you invest  $P$  dollars now at annual interest rate  $i$ . Take  $n$ ,  $i$ , and  $P$  as parameters. The formula is

$$A = P(1 + i)^n$$

- (b) Write a function to compute and return the amount of money,  $P$ , that you would need to invest now at annual interest rate  $i$  in order to have  $A$  dollars in  $n$  years. Take  $n$ ,  $i$ , and  $A$  as parameters. The formula is:

$$P = \frac{A}{(1 + i)^n}$$

- (c) Write a function that will read and validate the inputs for this program. Using call by value/address, return an enumerated constant for the choice of formulas, a number of years, an interest rate, and an amount of money, in dollars. All three numbers must be greater than 0.0.
- (d) Write a `main` program that will call the input routine to gather the data. Then, depending on the user's choice, it should call the appropriate calculation function and print the results of the calculation.

8. Sorting boxes.

A set of boxes are on the floor. We want to put them in two piles, those larger than the average box and those smaller than or equal to the average box. Write a program to label the boxes in the following manner:

- (a) Rewrite the `get_data()` function in Figure 11.14 so that you can enter data into three arrays rather than one. These arrays should hold the length, width, and height of the boxes, respectively.
- (b) Write a function, named `volume()`, to compute and store the volume of each box in a fourth parallel array. The volume of a box is the product of its length, width, and height.
- (c) Simplify the `stats()` function in Figure 11.14 so that it computes only the average, not the variance. Rename it `average()`. Then write a function, `print_boxes()`, that will print the data for each box in a nice table, including a column containing the appropriate label, `big` or `small`, depending on whether the volume of the box is larger or smaller than the average volume.

# Part IV

# Representing Data



# Chapter 12

## Strings

In this chapter, we begin to combine the previously studied data types into more complex units: strings and arrays of strings. Strings combine the features of arrays, characters, and pointers discussed in the preceding chapters. String input and output are presented, along with the most important functions from the standard —pl C++**string** library. Then strings and arrays are combined to form a data structure called a *ragged array*, which is useful in many contexts. Examples given here include a two applications with menu-driven user interfaces.

**Types and type names.** Variable names are like nouns in English; they are used to refer to objects. Type names are like adjectives; they are used to describe objects but are not objects themselves. A type is a pattern or rule for constructing future variables, and it tells us how those variables may be used. Types such as **int**, **float**, and **double** have names defined by the C standard. These are all simple types, meaning that objects of these types have only one part. We have also studied arrays, which are aggregate objects.

Other important types that we use all the time, such as **cstring**, do not have standard names in C; they must be defined by type specifications<sup>1</sup>. For example, although the C standard refers many times to strings, it does not define **string** as a type name; the actual type of a string in C is **char\***, or “pointer to character.”

### 12.1 String Representation

We have used literal strings as formats and output messages in virtually every program presented so far but have never formally defined a string type. We will use the typename **cstring** to refer to C-style strings. This is a necessary type and fundamental. C reference manuals talk about strings and the standard C libraries provide good support for working with strings. The functions in the **stdio** library make it possible to read and write strings in simple ways, and the functions in the **Cstring** library let you manipulate strings easily.

In contrast, C++ defines a type **string** and a **string** library of functions that are similar but not identical to the functions in the **Cstring** library. In this text, we will focus on the C++ strings and library. However, the C++ programmer must also understand C strings because they are still part of the language and are used for a variety of essential purposes.

Within C, the implementation of a **string** is not simple; it is defined as a pointer to a null-terminated array of characters. The goal in this section is to learn certain fundamental ways to use and manipulate strings and to learn enough about their representation to avoid making common errors. To streamline the coding, the following type definition will be used to define the type **cstring** as a synonym for a **const char\***.

The C++ **string** is a much more sophisticated type and is far easier to use. A C++**string** has a **Cstring** inside of it, plus additional information used for managing storage<sup>2</sup>. Happily, C++**strings** can be used without understanding how or why they work.

Suppose we want a string that contains different messages at different times. There are three ways to do this! First, it is possible to simply have an array of characters, each of which can be changed individually to form a new message. Second, the pointer portion of the **cstring** can be switched from one literal array of

---

<sup>1</sup>In contrast, **String** is a standard type in Java and **string** is standard in C++.

<sup>2</sup>This will be covered in Chapter 16.

---

A simple string literal:	<code>"George Washington\n"</code>
A two-part string literal:	<code>"was known for his unwavering honesty "</code>
One with escape codes:	<code>"and became our first president."</code>

---

**Figure 12.1.** String literals.

letters to another. Third, you can define a **C++ string** that can contain any sequence of letters, of any length. It will enlarge itself to store whatever you put in it. Also, the individual letters in a **C++ string** can be searched and changed, if needed.

These three string representations have very different properties and different applications, leading directly to programming choices, so we examine them more closely.

### 12.1.1 String Literals

A **string literal** is a series of characters enclosed in double quotes. String literals are used for output, as formats for `printf()` and `scanf()`, and as initializers for both **C strings** and **C++ strings**. Ordinary letters and escape code characters such as `\n` may be written between the quotes, as shown in the first line of Figure 12.1. In addition to the characters between the quotes, the C translator adds a null character, `\0`, to the end of every string literal when it is compiled into a program. Therefore, the literal `"cat"` actually occupies 4 bytes of memory and the literal `"$"` occupies 2.

**Two-part literals.** Often, prompts and formats are too long to fit on one line of code. In old versions of C, this constrained the way programs could be written. The ANSI C standard introduced the idea of *string merging* to fix this problem. With **string merging**, a long character string can be broken into two or more parts, each enclosed in quotes. These parts can be on the same line of code or on different lines, so long as there is nothing but whitespace or a comment between the closing quote of one part and the opening quote of the next. The compiler will join the parts into a single string with one null character at the end. An example of a two-line string literal is shown in the second part of Figure 12.1.

**Quotation marks.** You can even put a quote mark inside a string. To put a single quotation mark in a string, you can write it like any other character. To insert double quotes, you must write `\"`. Examples of quotes within a quoted string are shown in the last line of Figure 12.1.

**Strings and comments.** The relationship between comments and quoted strings can be puzzling. You can put a quoted phrase inside a comment and you can put something that looks like a comment inside a string. How can you tell which situation is which? The translator reads the lines of your source code from top to bottom and left to right. The symbol that occurs first determines whether the following code is interpreted as a comment or a string. If a begin-comment mark is found after an open quote, the comment mark and the following text become part of the string. On the other hand, if the `//` or `/*` occurs first, followed by an open quote, the following string becomes part of the comment. In either case, if you attempt to put the closing marks out of order, you will generate a compiler error. Examples of these cases are shown in Figure 12.2.

---

A comment in a string:	<code>"This // is not a comment it is a string."</code>
A string in a comment:	<code>// Here's how to put "a quote" in a comment.</code>
Overlap error:	<code>"Here /* are some illegal" misnested delimiters. */</code>
Overlap error:	<code>/* We can't "misnest things */ this way, either."</code>

---

**Figure 12.2.** Quotes and comments.

- The null character is 1 byte (type `char`) filled with 0 bits.
- The null pointer is a pointer filled with 0 bits (it points to address 0). In C it is called `NULL`. In C++ it is called `nullptr`.
- The null string is a pointer to a null character. The address in the pointer is the nonzero address of some byte filled with 0 bits.

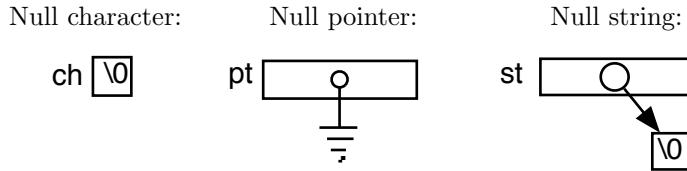


Figure 12.3. Three kinds of nothing.

### 12.1.2 A String Is a Pointer to an Array

We have said that a `cstring` is a series of characters enclosed in double quotes and that a `cstring` is a pointer to an array of characters that ends with the null character. Actually, both these seemingly contradictory statements are correct: When we write a string in a program using double quotes, it is translated by the compiler into a pointer to an array of characters.

**String pointers.** One way to create a string in C is to declare a variable of type `cstring` or `char*` and initialize it to point at a string literal.<sup>3</sup> Figure 12.4 illustrates a **cstring variable** named `word`, which contains the address of the first character in the sequence of letters that makes up the literal "`Holiday`". The pointer variable and the letter sequence are stored in different parts of memory. The pointer is in the user's memory area and can be changed, as needed, to point to a different message. In contrast, the memory area for literals cannot be altered. C and C++ forbid us to change the letters of "`Holiday`" to make it into "`Ponyday`" or "`Holyman`".

**Null objects.** The null character, the null pointer, and the null string (pictured in Figure 12.3) often are confused with each other, even though they are distinct objects and used in different situations.

The **null string** is a string with no characters in it except the null character. It is denoted in a program by two adjacent double quote marks, `""`. It is a legal string and often useful as a placeholder or initializer. Both the null pointer and the null string are pointers, but the first is all 0 bits, the address of machine location 0, and the second is the nonzero address of a byte containing all 0 bits.

The **null character** (eight 0 bits), or **null terminator** is used to mark the end of every string and plays a central role in all string processing. When you read a `cstring` with `scanf()`, the null character automatically is appended to it for you. When you print a `cstring` with `printf()` or `puts()`, the null character marks where to stop printing. The functions in the C and C++ `string` libraries all use the null character to terminate their loops. See Figure 12.3.

**Using a pointer to get a string variable.** We can use a `char*` variable and **string assignment** to select one of a set of predefined messages and remember it for later processing and output. The `char*` variable acts like a finger that can be switched back and forth among a set of possible literals.

For example, suppose you are writing a simple drill-and-practice program and the correct answer to a question, an integer, is stored in `target_answer` (an integer variable). Assume that the user's input has been read into `input_answer`. The output is supposed to be either the message "Good job!" when a correct answer is entered or "Sorry!" for a wrong answer. Let `response` be a string variable, as shown in Figure 12.5. We can

<sup>3</sup>Dynamically allocated memory also can be used as an initializer. This advanced topic is left for Chapter 16.

```
typedef char* cstring;
cstring word = "Holiday";
```

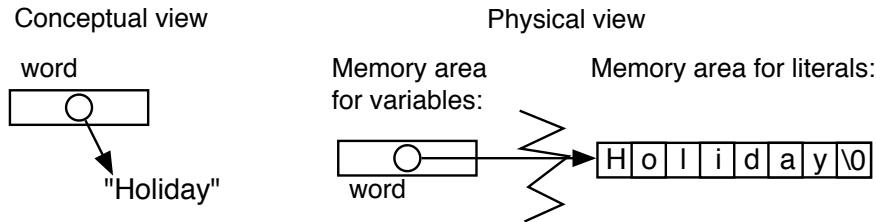


Figure 12.4. Implementation of a `cstring` variable.

select and remember the correct message by using an `if` statement and two pointer assignment statements, as shown.

### 12.1.3 Declare an Array to Get a String

To create a `cstring` whose individual letters can be changed, as when a word is entered from the keyboard, we must create a character array big enough to store the longest possible word. Figure 12.6 shows one way to do this with an array declaration.<sup>4</sup> Here, the variable `president` is a character array that can hold up to 17 letters and a null terminator, although only part of this space is used in this example.

**Initializers.** A character array can be declared with or without an initializer. This initializer can be either a quoted string or a series of character literals, separated by commas, enclosed in braces. The array `president` in Figure 12.6 is initialized with a quoted string. The initializer also could be

```
= {'A', 'b', 'r', 'a', 'h', 'm', ' ', 'L', 'i', 'n', 'c', 'o', 'l', 'n', '\0'}
```

This kind of initializer is tedious and error prone to write, and you must remember to include the null character at the end of it. Because they are much more convenient to read and write, quoted strings (rather than lists of literal letters) usually are used to initialize `char` arrays. The programmer must remember, though, that every string ends in a null character, and there must be extra space in the array for the `\0`.

**Using character arrays.** If you define a variable to be an array of characters, you may process those characters using subscripts like any other array. Unlike the memory for a literal, this memory can be updated.

It also is true that an array name, when used in C with no subscripts, is translated as a pointer to the first slot of the array. Although an array is not the same as a string, this representation allows the name of a

<sup>4</sup>The use of dynamic allocation for this task will be covered in Chapter 16.

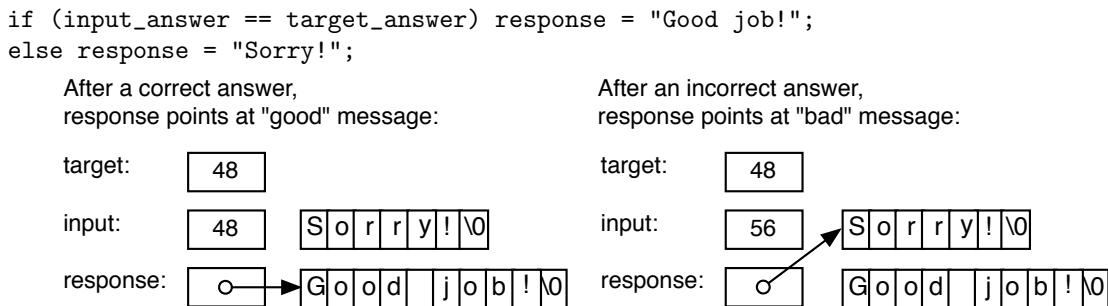
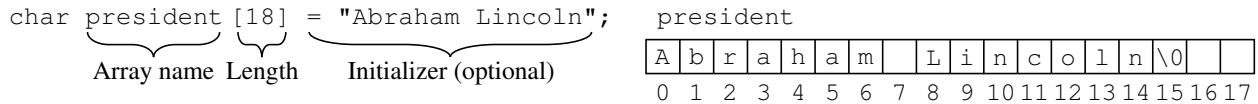


Figure 12.5. Selecting the appropriate answer.

---

An array of 18 `chars` containing the letters of a 15-character string and a null terminator. Two array slots remain empty.



**Figure 12.6. An array of characters.**

character array to be *used* as a string (a `char*`). Because of this peculiarity in the semantics of C, operations defined for strings also work with character arrays. Therefore, in Figure 12.6, we could write `puts( president )` and see `Abraham Lincoln` displayed.

However, an array that does not contain a null terminator must not be used in place of a string. The functions in the `string` library would accept it as a valid argument; the system would not identify a type or syntax error. But it would be a semantic error; the result is meaningless because C will know where the string starts but not where it ends. Everything in the memory locations following the string will be taken as part of it until a byte is encountered that happens to contain a 0.

#### 12.1.4 Array vs. String

Typical uses of strings are summarized in Figure 12.7. Much (but not all) of the time, you can use strings without worrying about pointers, addresses, arrays, and null characters. But often confusion develops about the difference between a `char*` and an array of characters. A common error is to declare a `char*` variable and expect it to be able to store an array of characters. However, a `char*` variable is only a pointer, which usually is just 4 bytes long. It cannot possibly hold an entire word like `"Hello"`. If we want a string to store alphabetic data, we need to allocate memory for an array of characters. These issues can all be addressed by understanding how strings are implemented in C.

Figure 12.8 shows declarations and diagrams for a character array and a `char*` variable, both initialized by quoted strings. As you can see, the way storage is allocated for the two is quite different. When you use an array as a container for a string, space for the array is allocated with your other variables, and the letters of an initializing string (including the null) are copied into the array starting at slot 0. Any unused slots at the end of the array are left untouched and therefore contain garbage. In contrast, two separate storage areas are used when you use a string literal to initialize a pointer variable such as `str`. A string pointer typically occupies 4 bytes while the information itself occupies 1 byte for each character, plus another for the null character. The compiler stores the characters and the null terminator in an unchangeable storage area separate from the declared variables, and then it stores the address of the first character in the pointer variable.

---

Use type `char*` for these purposes:

- To point at one of a set of literal strings.
- As the type of a function parameter where the argument is either a `char*` or a `char` array.

Use a `char` array for these purposes:

- To hold keyboard input.
- Any time you wish to change some of the letters individually in the string.

---

**Figure 12.7. Array vs. string.**

These declarations create and initialize the objects diagrammed below:

```
char ara[7] = "Monday"; // 7 bytes total.
char* str = "Sunday"; // 11 bytes total, 4 for pointer, 7 for chars.
char* str2 = &str[3]; // 4 more bytes for the pointer.
```

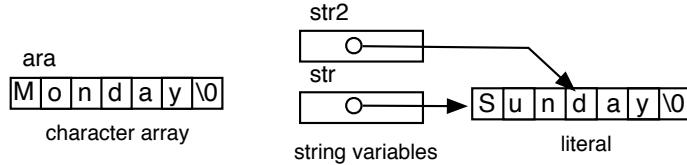


Figure 12.8. A character array and a string.

### 12.1.5 C++ Strings

A C++ string contains a C string plus memory management information. The array part of the string is dynamically allocated, and the length of the allocation is stored as part of the object. (This is labelled “max” in Figure 12.9) The other integer, labelled “len”, is the actual length of the string.

When a variable is created and initialized to a string value, enough space is allocated to store the value plus a null terminator, plus padding bytes to make the length of the entire allocation a power of 2 (8, 16, 32 bytes, etc.)<sup>5</sup>. If the string becomes full and more space is needed, the initial space is doubled. In this way, a C++ string variable can hold a string of any length: it “grows” when needed to hold more characters.

Because of the automatic resizing, C++ strings are strongly preferred for all situations in which the length of the string is unknown or not limited.

## 12.2 C String I/O

Even though the string data type is not built into the C language, functions in the standard libraries perform input, output, and other useful operations with strings. Several code examples are given in this chapter. Because it is important to know how to do input and output in both languages, these examples are given first in C, then in C++.

### 12.2.1 String Output

We have seen two ways to output a string literal in C: `puts()` and `printf()`. The `puts()` function is called with one string argument, which can be either a literal string, the name of a string variable, or the name of a character array. It prints the characters of its argument up to, but not including, the null terminator, then adds a newline (\n) character at the end. If the string already ends in \n, this effectively prints a blank line following the text. We can call `printf()` in the same way, with one string argument, and the result will be the

<sup>5</sup>This is a simplified explanation. Modern compilers also have optimized representations for short strings.

These declarations create the C++ string objects diagrammed below.

```
string s1 = "Monday";
string s2("Tuesday");
```

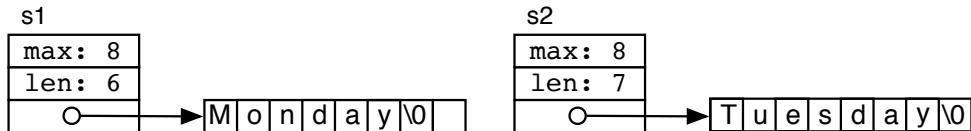


Figure 12.9. C++ strings.

```
#include <stdio.h>
int main( void )
{
    char* s1 = "String demo program.";
    char* s2 = "Print this string and ring the bell.\n";
    puts( s1 );
    printf( "\t This prints single ' and double \" quotes.\n" );
    printf( "\t %s\n%c", s2, '\a' );

    puts( "These two quoted sections " "form a single string.\n" );
    printf( "You can break a format string into pieces if you\n"
            "end each line and begin the next with quotes.\n"
            "You may indent the lines any way you wish.\n\n" );

    puts( "This\tstring\thas\ttab\tcharacters\tbetween\twords.\t" );
    puts( "Each word on the line above starts at a tab stop. \n" );
    puts( "This puts()\n\t will make 3 lines of output,\n\t indented.\n" );

    printf( " >>%-35s<< \n >>%35s<< \n",
            "left justify, -field width", "right justify, +field width" );
    printf( " >>%10s<< \n\n", "not enough room? Field gets expanded" );
}
```

Figure 12.10. String literals and string output in C.

same except that a newline will not be added at the end. The first box in Figure 12.10 demonstrates calls on both these functions with single arguments.

**String output with a format.** We can also use `printf()` with a `%s` conversion specifier in its format to write a string. The string still must have a null terminator. Again, the corresponding string argument may be a literal, string variable, or `char` array. Examples include

```
string fname = "Henry";
printf( "First name: %s\n", fname );
printf( "Last name: %s\n", "James" );
```

The output from these statements is:

```
First name: Henry
Last name: James
```

Further formatting is possible. A `%ns` field specifier can be used to write the string in a fixed-width column. If the string is shorter than `n` characters, spaces will be added to fill the field. When `n` is positive, the spaces are added on the left; that is, the string is **right justified** in a field of `n` columns. If `n` is negative, the string is left justified. If the string is longer than `n` characters, the entire string is printed and extends beyond the specified field width. In the following example, assume that `fname` is "Andrew":

```
printf( ">>%10s<<\n", fname );
printf( ">>%-10s<<\n", fname );\\
```

Then the output would be

```
>> Andrew<<
>>Andrew <<
```

```

1 // -----
2 // Figure 12.11: String literals and string output in C++.
3 // Rewritten in C++ June 21, 2016
4 // -----
5 #include "tools.hpp"
6 typedef const char* cstring;
7
8 int main( void )
9 {
10     cstring s1 = "\nString demo program.";
11     cstring s2 = "Print this string and a newline.\n";
12
13     cout << s1 << endl;
14     cout << s2;
15     cout << "\t This line prints single ' and double \" quote marks.\n\n" ;
16
17     cout << "These two quoted sections " "form a single string literal.\n" ;
18     cout << "You can break a format string onto several lines if you end \n"
19         "each line with a quote and begin the next line with a quote.\n"
20         "You may indent the lines any way you wish.\n\n";
21
22     cout << "This\tstring\thas\ta\ttab\tcharacter\tafter\tteach\tword.\n";
23     cout << "Each word on the line above should start at a tab stop. \n";
24     cout << "This line\n \t will make 3 lines of output,\n \t indented.\n\n";
25
26     cout <<"[" <<left <<setw(35) <<"A left-justified string" <<">]\n";
27     cout <<"[" <<right <<setw(35) <<"and a right-justified string" <<">]\n";
28     cout <<"[" <<left <<setw(35) <<"Not enough room? Field gets expanded." <<">]\n";
29     cout <<endl;
30 }
31
32 /* Output: -----
33
34 String demo program.
35 Print this string and a newline.
36     This line prints single ' and double " quote marks.
37
38 These two quoted sections form a single string literal.
39 You can break a format string onto several lines if you end
40 each line with a quote and begin the next line with a quote.
41 You may indent the lines any way you wish.
42
43 This string has a tab character after each word.
44 Each word on the line above should start at a tab stop.
45 This line
46     will make 3 lines of output,
47     indented.
48
49 [[A left-justified string          ]]
50 [[      and a right-justified string]]
51 [[Not enough room? Field gets expanded.]]
52
53 */

```

---

Figure 12.11. String literals and string output in C++.

**Notes on Figure 12.10. String literals and string output.** In this program, we demonstrate a collection of string-printing possibilities.

**First box: ways to print strings.**

- On the first line, we use `puts()` to print the contents of a string variable.
- The second line calls `printf()` to print a literal string. When printing just one string with `printf()`, that string becomes the format, which always is printed. This line also shows how escape characters are used to print quotation marks in the output.
- The third line prints both a string and a character. The `%s` in the format prints the string "Print this string and ring the bell." and then goes to a new line; the `%c` prints the character code `\a`, which results in a beep with no visible output on our system.
- The output from this box is

```
String demo program.  
This prints single ' and double " quotes.  
Print this string and ring the bell.
```

- In C++ there is only one way to output a string: the operator `<<`. Figure 12.11, shows the same demo program translated to C++. Lines 13–15 correspond to Box 1 of the C program. The output from the C++ version is shown below the code.

**Second box: adjacent strings are united.**

- The code in Box 2 corresponds to lines 17–20 of the C++ version.
- The `puts()` shows that two strings, separated only by whitespace, become one string in the eyes of the compiler. This is a very useful technique for making programs more readable. Whitespace refers to any nonprinting character that is legal in a C source code file. This includes ordinary space, newline, vertical and horizontal tabs, formfeed, and comments. If we were to write a comma between the two strings, only the first one would be used as the format, the second one would cause a compiler error.
- The output from this box is

```
These two quoted sections form a single string.  
  
You can break a format string into pieces if you  
end each line and begin the next with quotes.  
You may indent the lines any way you wish.
```

- When we have a long format in a `printf()` statement, we just break it into two parts, being sure to start and end each part with a double quote mark. Where we break the format string and how we indent it have no effect on the output.

**Third box: escape characters in strings.**

- The code in Box 3 corresponds to lines 22–24 of the C++ version.
- The output from this box is shown below a tab line so you can see the reason for the unusual spacing:

--	--	--	--	--	--	--

```
This string has tab characters between words.  
Each word on the line above starts at a tab stop.  
  
This puts()  
will make 3 lines of output,  
indented.
```

- Note that the tab character does not insert a constant number of spaces, it moves the cursor to the next tab position. Where the tab stops occur is defined by the software (an eight-character stop was used here). In practice, it is a little tricky to get things lined up using tabs.
- The presence or absence of newline characters controls the vertical spacing. The number of lines of output does not correspond, in general, to the number of lines of code.

Both these `scanf()` statements are errors. The first one stores the input at some random memory location. The second one will not compile because it is attempting to change a string literal.

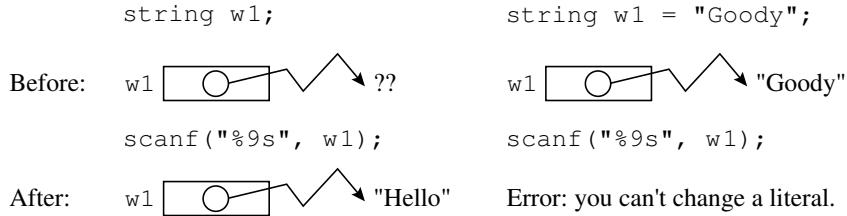


Figure 12.12. You cannot store an input string in a pointer.

#### **Fourth box: using field-width specifiers with strings.**

- The output is

```

>>left justify, -field width      <<
>>          right justify, +field width<<
>>not enough room? Field gets expanded<<
  
```

- Spaces are used to pad the string if it is shorter than the specified field width. Using a negative number as a field width causes the output to be left justified. A positive field width produces right justification.
- If the string is longer than the field width, the width specification is ignored and the entire string is printed.
- Formatting in C++ is very different from C because the output operator does not use formats. Instead, syntactic units called “manipulators” are written in the output stream, as if they were values to print. Lines 26–28 show how to set the field width and justification (right or left).

Line 29 uses `endl`. This manipulator prints a newline and flushes the output from the internal buffer to the screen.

### 12.2.2 String Input in C

String input is more complicated than string output. The string input functions in the C standard I/O library offer a bewildering variety of options; only a few of the most useful are illustrated here. One thing common to all C string input functions is that they take one string argument, which should be the name of a character array or a pointer to a character array. *Do not use a variable of type `char*` for this purpose unless you have previously initialized it to point at a character array.*

For example, the diagram in Figure 12.12 shows two before-and-after scenarios that cause trouble. The code fragment on the left is wrong because the pointer is uninitialized, so it points at some random memory location; attempting to store data there is likely to cause the program to crash. The code on the right is wrong because C does not permit a program to change a literal string value. The diagrams in Figure 12.13 show two ways to do the job right. That on the left uses the name of an array in the `scanf()` statement; that on the right uses a pointer initialized to point at an array.

Also, an array that receives string input must be long enough to contain all the information that will be read from the keyboard plus a null terminator. The input, once started by the user hitting the Enter key, continues until one of two conditions occurs:

1. The input function reads its own termination character. This varies, as described next, according to the input function.
2. The maximum number of characters is read, as specified by some field-width limit.

After the read operation is over, a null character will be appended to the end of the data.

**Reading a string with `scanf()`.** The easiest way to read a string is by using `scanf()` with a **%s conversion specifier**. This will skip over leading whitespace characters, then read and store input characters at the address given, until whitespace is encountered in the input. Thus, `%s` will read in only one word.

To avoid security problems, a number (the length of your array -1) must be written between the `%` and the `s` in your format. If the read operation does not stop sooner because of whitespace in the input stream, it will stop after reading `n` input characters and store a null terminator after the `n`th char. For example, `%9s` will stop

Here are two correct and meaningful ways to read a string into memory:

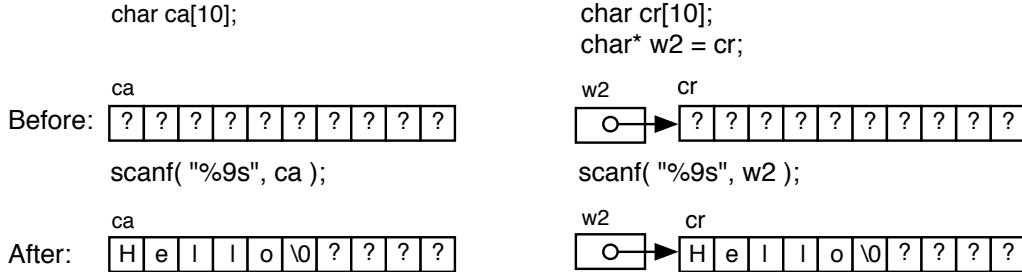


Figure 12.13. You need an array to store an input string.

the read operation after nine characters are read; the corresponding array variable must be at least 10 bytes long to allow enough space for the input plus the null terminator. Here are examples of calls on `scanf()` with and without a field-width limit. The correct call is first, the faulty one second.

```
char name[10];
scanf( "%9s", name );    // safe
scanf( "%s", name );    // dangerous; overflow possible. DO NOT do this.
```

Note that there is no ampersand before the variable name; the `&` must be omitted because `name` is an array.

**The brackets conversion.** If the programmer wants some character other than whitespace to end the `scanf()` read operation, a format of the form `%n[^?]` may be used instead of `%ns` (`n` is still an integer denoting the field length). The desired termination character replaces the question mark and is written inside the brackets following the carat character, `^`.<sup>6</sup> Some examples:

```
scanf( "%29[^\\n]", street );
scanf( "% 29[^,], %2[^\\n]", city, state );
```

A complete input would be:

```
122 E. 42nd St.  
New York, NY
```

The first line reads the street address and stops the read operation after 29 characters have been read, or sooner if a newline character is read. It does not stop for other kinds of whitespace. The second example reads characters into the array `city` until a comma is read. It then reads and discards the comma and reads the remaining characters into the array `state` until a newline is entered.

**Formats for `scanf()`.** An input format contains a series of conversion specifiers. It can also contain blanks, which tell `scanf()` to ignore whitespace characters in that position of the input. Anything in an input format that is not a blank or part of a conversion specifier *must* be present in the input for correct operation. At run time, when that part of the format is interpreted, characters are read from the input buffer, compared to the characters in the format, and discarded if they match. If they do not match, it is an input error and they are left in the buffer<sup>7</sup>.

In the second example above, the program reads two strings (city and state) that are on one input line, separated by a comma and one or more spaces. The comma is detected first by the brackets specifier but not removed from the buffer. The comma and spaces then are read and removed from the buffer because of the `", "` in the format. Finally, the second string is read and stored. These comma and whitespace characters are discarded, not stored in a variable. Whenever a non-whitespace character is used to stop an input operation, that character must be removed from the buffer by including it directly in the format or through some other method. Figure 12.14, in the next section, shows these calls on `scanf()` in the context of a complete program.

<sup>6</sup>A variety of effects can be implemented using square brackets. We present only the most useful here. The interested programmer should consult a standard reference manual for a complete description.

<sup>7</sup>This is another error condition that will be discussed in Chapter 14.

**The `gets()` function** C supports another string input function called `gets()` (get-string) with the prototype `string gets( char s[] )`. It reads characters (including leading whitespace) from the input stream into the array until the first `\n` is read. The newline character terminates reading but is *not* stored in `s`. Instead, a null terminator is stored at the end of the input characters in the array. The variable `s` must refer to an array of `char`s long enough to hold the entire string and the null character.

```
char course_name[20];
gets( course_name ); // Error if input > 19 chars.
```

At first sight, it would seem that `gets()` is a good tool; it is simple and it reads a whole line without the fuss of making a format. However, this function has a fatal flaw that makes it inappropriate for use by responsible programmers: there is no way to make the read operation stop when it reaches the end of the array. If the input is too long, it will simply overwrite memory and wipe out the contents of adjacent memory variables. We call this “*walking on memory*”, and it has been the basis of countless viruses and attacks. For this reason, responsible programmers avoid `gets()` and we do not deal further with it in this text.

### 12.2.3 String Input in C++

The input operator in C++ is `>>`. To input something from the keyboard, we use `cin >>`. These are the most important things to know about input using `cin >>`:

- The type of value read is determined by the type of the variable you read it into. No format is needed.
- Built-in input methods exist for all the numeric types, characters, and strings.
- When using `>>` to read something, leading whitespace will be skipped by default<sup>8</sup>.
- A string input ends when the first whitespace happens. You cannot read an entire name or sentence this way.
- When reading strings with `>>`, it is possible to read more characters than will fit into a `char` array. For this reason, it is *unsafe and unprofessional* to read a string with `>>`.

Thus, we need some other safe way to read strings in C++. The library provides three functions (with multiple versions) for this task. One is named `get()` and the other two named `getline()`. None of them skip leading whitespace, and all of them put a null terminator on the end of the string that was read in.

- **`getline( cin, stringVar )`:**  
This function is used to read a string of any length into a C++ string variable. The variable will “grow”, as needed, to store a string of any length. The read operation stops when the next newline character is entered.
- **`getline( cin, stringVar, charDelim )`:**  
When a third parameter is used to call `getline()`, it is used instead of `\n` as the delimiter to end of the input. This allows you to read an entire file in one line.
- **`cin.getline( charArray, maxSize )`:**  
Characters are read from `cin` and stored in the array until the newline character is found or the `maxSize-1` has been reached. A null terminator is stored in the last position. The newline is removed from the stream and discarded. It is not stored in the array.
- **`cin.getline( charArray, maxSize, charDelim )`:** Characters are read from `cin` and stored in the array until the delimiter character is found or the `maxSize-1` has been reached. A null terminator is stored in the last position. The delimiter is removed from the stream and discarded. It is not stored in the array.
- **`cin.get( charArray, maxSize ) and cin.get( charArray, maxSize, charDelim )`:**  
This function is exactly like `getline()` except that it leaves the delimiter in the stream instead of discarding it. That char must be handled by some future read operation.

---

<sup>8</sup>The default can be changed but that is uncommon.

### 12.2.4 Guidance on Input

In general, you should use a C++ `string` and `getline( cin, stringVar )` for all string input. One of the plagues of mankind is malware, and many of the vulnerabilities that make malware possible are caused by inappropriate string input operations that can overflow their arrays and start overlaying adjacent memory. Using C++ `string` and `getline( cin, stringVar )` avoids having this kind of vulnerability in your code. It is also a great deal easier than using arrays and pointers.

## 12.3 String Functions

The C and C++ libraries provide a library of functions that process strings. In addition, programmers can define their own string functions. In this section, we first examine how strings are passed as arguments and used as parameters in functions. Then we explore the many built-in string functions and how they can be used in text processing.

In the C++ code examples, you will note that functions are called in the OO manner, by putting the name of an object (and a dot) before the name of the function.

### 12.3.1 Strings as Parameters

The program example in Figure 12.14 demonstrates the proper syntax for a function prototype, definition, and call with a string parameter. Figure 12.15 repeats the same demonstration in C++.

**Notes on Figure 12.14. A string parameter.**

*First box: string input.*

1. In the C++ version, lines 15..24 correspond to Box 1.
2. The safe methods of string input described in the previous section are used here in a complete program. In every case, a call on `scanf()` limits the number of characters read to one less than the length of the array that receives them, leaving one space for the null terminator.
3. In the first call, the argument to `scanf()` is a string variable initialized to point to a character array. It is the correct type for input using either the `%s` or `%[^?]` conversion specifiers.
4. In the other two calls, the argument is a character array. This also is a correct type for input using these specifiers.

*Second box: character and string output.*

1. In the C++ version, lines 26–30 correspond to Box 2.
2. Both `putchar()` and `puts()` are used to print a newline character. The arguments are different because these functions have different types of parameters. The argument for `putchar()` is the single character, `\n`, while the string "`\n`" is sent to `puts()`, which prints the newline character and adds another newline.
3. Inner boxes: calls on `print_upper()`. The programmer-defined function `print_upper()` is called twice to print upper-case versions of two of the input strings. In the first call, the argument is a string variable. In the second, it is a character array. Both are appropriate arguments when the parameter has type `string`. We also could call this function with a literal string.
4. Sample output from this program might be

```
Enter first name: Maggie
Enter street address: 180 River Rd.
Enter city, state: Haddon, Ct

MAGGIE
180 River Rd. Haddon, CT
```

*Last box: the `print_upper()` function.*

1. In the C++ version, lines 35–39 correspond to Box 1.
2. The parameter here is type `string`. The corresponding argument can be the name of a character array, a literal string, or a string variable. It could also be the address of a character somewhere in the middle of a null-terminated sequence of characters.

3. Since a string is a pointer to an array of characters, those characters can be accessed by writing subscripts after the name of the string parameter, as done here. The `for` loop starts at the first character of the string, converts each character to upper case, and prints it. The original string remains unchanged.
4. Like most loops that process the characters of a string, this loop ends at the null terminator.

### 12.3.2 The String Library

The standard C **string library** contains many functions that manipulate strings. Some of them are beyond the scope of this book, but others are so basic that they deserve mention here:

- `strlen()` finds the length of a string.
- `strcmp()` compares two strings.
- `strncmp()` compares up to  $n$  characters of two strings.
- `strcpy()` copies a whole string.

This program demonstrates a programmer-defined function with a string parameter. It also uses several of the string input and output functions in a program context.

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

void print_upper( char* s );

int main( void )
{
    char first[15];
    char* name = first;
    char street[30], city[30], state[3];

    printf( " Enter first name: " );
    scanf( "%14s", name );           // Read one word only
    printf( " Enter street address: " );
    scanf( "%29[^\\n]", street );    // Read to end of line
    printf(" Enter city, state: " );   // Split line at comma
    scanf( "%29[^,], %2[^\\n]", city, state );

    putchar( '\\n' );
    print_upper( name );           // Print name in all capitals.
    printf( "\\n %s %s, ", street, city ); // Print street, city as entered.
    print_upper( state );          // Print state in all capitals.
    puts( "\\n" );
}

// ----- Print letters of string in upper case.
void print_upper( char* s )
{
    for (int k = 0; s[k] != '\\0'; ++k) putchar( toupper( s[k] ) );
}
```

Figure 12.14. A string parameter in C.

```
1 // -----
2 // Figure 12.15: A string parameter.
3 // Rewritten in C++ June 21, 2016
4 // -----
5 #include "tools.hpp"
6
7 void printUpper( string s );
8
9 int main( void )
10 {
11     string name;
12     string street;
13     char city[30], state[3];
14
15     cout <<"\n Enter first name: ";
16     cin >>name;                                // Read one word only; space delimited.
17     cout <<" Enter street address: ";
18     cin >> ws;
19     getline(cin, street );                     // Read to end of line
20     cout <<" Enter city, state: ";
21     cin >> ws;
22     cin.getline( city, 30, ',' );              // Split line at comma
23     cin >> ws;
24     cin.getline( state, 3 );
25
26     cout <<"\n\n";
27     printUpper( name );                      // Print name in all capitals.
28     cout << " " <<street <<, " <<city <<, " ; // Print street, city as entered.
29     printUpper( state );                     // Print state in all capitals.
30     cout <<endl;
31     return 0;
32 }
33
34 // -----
35 void printUpper( string s )
36 {
37     for( int k = 0; s[k] != '\0'; ++k ) s[k] = toupper(s[k]);
38     cout << s << endl;
39 }
40
41 /* Output: -----
42
43 Enter first name: janet
44 Enter street address: 945 Center St.
45 Enter city, state: Houghton, wv
46
47
48 JANET
49      945 Center St., Houghton, WV
50
51 ----- */
```

---

Figure 12.15. A string parameter in C++.

```
#include <stdio.h>
#include <cstring.h>
#define N 5

int main( void )
{
    char* w1 = "Annette";
    char* w2 = "Ann";
    char w3[20] = "Zeke";

    printf( " sizeof w1 is %2i   string is \"%s\"\t strlen is %i\n",
            sizeof w1, w1, strlen( w1 ) );
    printf( " sizeof w2 is %2i   string is \"%s\"\t\t strlen is %i\n",
            sizeof w2, w2, strlen( w2 ) );
    printf( " sizeof w3 is %2i   string is \"%s\"\t\t strlen is %i\n\n",
            sizeof w3, w3, strlen( w3 ) );
}
```

The output of this program is

```
sizeof w1 is 4   string is "Annette"   strlen is 7
sizeof w2 is 4   string is "Ann"       strlen is 3
sizeof w3 is 20  string is "Zeke"     strlen is 4
```

---

**Figure 12.16.** The size of a C string.

- **strncpy()** copies up to  $n$  characters of a string. If no null terminator is among these  $n$  characters, do not add one.
- **strcat()** copies a string onto the end of another string.
- **strncat()** copies up to  $n$  characters of a string onto the end of another string. If no null terminator is among these  $n$  characters, add one at the end.
- **strchr()** finds the leftmost occurrence of a given character in a string.
- **strrchr()** finds the rightmost occurrence of a given character in a string.
- **strstr()** finds the leftmost occurrence of a given substring in a string.

We discuss how these functions can be used correctly in a variety of situations and, for some, show how they might be implemented.

**The length of a string.** Because a string is defined as a **two-part object** (recall the diagrams in Figure 12.8), we need two methods to find the size of each part. Figures 12.16 and 12.17 illustrate the use of both operations. The operator **sizeof** returns the size of the pointer part of a string in C and the size of the string object in C++, while the function **strlen()** returns the number of characters in the array part, not including the null terminator. (Therefore, the string length of the null (empty) string is 0.) Figure 12.19 shows how **strlen()** might be implemented. Figure 12.29 shows further uses of **strlen()** in a program.

#### Notes on Figure 12.16. The size of a string.

1. The variables **w1** and **w2** are strings. When the program applies **sizeof** to a string, the result is the size of a pointer on the local system. So **sizeof w1 == sizeof w2** even though **w1** points at a longer name than **w2**.
2. In contrast, **w3** is not a string; it is a **char** array used as a container for a string. When the programmer prints **sizeof w3**, we see that **w3** occupies 20 bytes, which agrees with its declaration. This is true even though only 5 of those bytes contain letters from the string. (Remember that the null terminator takes up 1 byte, even though you do not write it or see it when you print the string.)

---

```

1 // -----
2 // Figure 12.17: The size of a C++ string.
3 // Rewritten in C++ June 21, 2016
4 //
5 #include "tools.hpp"
6 #define N 5
7
8 int main( void )
9 {
10     string w4( "Annabel" );
11
12     cout << " sizeof w4 is " << sizeof w4 << " --- content is " << w4
13         << " --- length is " << w4.length();
14     cout << "\n\n Can you explain why?\n\n" ;
15 }
16
17 /* Output: -----
18
19    sizeof w4 is 24 --- content is Annabel --- length is 7
20
21    Can you explain why?
22 */

```

---

**Figure 12.17. The size of a C++ string.**

3. To find the number of letters in a string, we use `strlen()`, string length, not `sizeof`. The argument to `strlen()` can be a `string` variable, a string literal, or the name of a character array. The value returned is the number of characters in the array part of the string, up to but not including the null terminator.
4. In C++, `sizeof` works exactly as it does in C. However, to find the number of letters in a string, we use `strName.length()`.

**Comparing two strings.** Figure 12.18 shows what happens when we try to compare two strings or `char` arrays using `==`. Unfortunately, only the addresses get compared. So if two pointers point at the same array, as with `w5` and `w6`, they are equal; if they point at different arrays, as with `w4` and `w5`, they are not equal, even if the arrays contain the same characters.

To overcome this difficulty, the C `string` library contains the function `strcmp()`, which compares the actual characters, not the pointers. It takes two arguments that are strings (pointers) and does a letter-by-letter, alphabetic-order comparison. The return value is a negative number if the first string comes before the other alphabetically, a 0 if the strings are identical up through the null characters, and a positive number if the second string comes first. We can use `strcmp()` in the test portion of an `if` or `while` statement if we are careful. Because the intuitively opposite value of 0 is returned when the strings are the same, we must remember to compare the result of `strcmp()` to 0 in a test for equality. Figure ?? shows a proper test for equality and Figure 12.20 shows how this function could be implemented.

The other standard string comparison function, `strncmp()`, also deserves mention. A call takes three arguments and has the form `strncmp( s1, s2, n )`. The first two parameters are just like `strcmp()`; the third parameter is used to limit the number of letters that will be compared. For example, if we write `strncmp( "elephant", "elevator", 3 )`, the result will be 0, meaning that the strings are equal up to the 3rd character. If `n` is greater than the length of the strings, `strncmp()`, works identically to `strcmp()`.

**Character search.** The function `strchr( string s1, char ch )` searches `s1` for the first (leftmost) occurrence of `ch` and returns a pointer to that occurrence. If the character does not appear at all, `NULL` is returned. The function `strrchr( string s1, char ch )` is similar, but it searches `s1` for the last (rightmost) occurrence of `ch` and returns a pointer to that occurrence. If the character does not appear at all, `NULL` is returned. Figure 12.21 shows how to call both these functions and interpret their results.

---

```

char w1[6] = "Hello";
cstring w2 = "Hello";

w1 [H e l l o \0]
w2 [O ] → "Hello"

w3
max: 8
len: 5
O → [H e l l o \0] [ ]
string w3 = w2;
string w4 = w3;

w4
max: 8
len: 5
O → [H e l l o \0] [ ]

```

---

```

1 // -----
2 // Figure 12.18: Do not use == with C strings.
3 // Rewritten in C++ June 21, 2016
4 // This demo defines what == means.
5 // -----
6 #include "tools.hpp"
7 #define N 5
8 typedef const char* cstring;
9
10 int main( void )
11 {
12     char w1[6] = "Hello";
13     cstring w2 = "Hello";
14     string w3 = w2;
15     string w4 = w3;
16
17     if (w1 == w2)    cout << "Yes, w1 == w2";
18     else            cout << "No, w1 != w2";
19     cout << " because we are comparing the pointers here." << endl;
20
21     if (strcmp( w1, w2 ) == 0)  cout << "Yes, w1 str= w2";
22     else                      cout << "No, w1 not str= w2";
23     cout << " because we are comparing the characters." << endl;
24
25     if (w2 == w3)    cout << "Yes, w2 == w3";
26     else            cout << "No, w2 != w3";
27     cout << " because they both point at the same thing." << endl;
28
29     if ( w3.compare(w4) )      cout << "w3 compares!= to w4";
30     else                      cout << "w3 compares== to w4";
31     cout << " because w4 is a copy of w3.\n\n";
32 }
33 /* Output: -----
34
35 No, w1 != w2 because we are comparing the pointers here.
36 Yes, w1 str= w2 because we are comparing the characters.
37 Yes, w2 == w3 because they both point at the same thing.
38 w3 compares== to w4 because w4 is a copy of w3.
39
40 */

```

---

Figure 12.18. Do not use == with C strings (but it works in C++).

---

Walk down a string, from the beginning to the null terminator, counting characters as you go. Do not count the terminal null character.

```
int my_strlen( char * st )      // One way to calculate the string length.
{
    int k;                      // Loop counter and return value.
    for (k=0; st[k]; ++k);     // A tight loop -- no body needed.
    return k;
}
```

---

Figure 12.19. Computing the length of a string in C.

**Substring search.** The function `strstr( string s1, string s2 )` searches `s1` for the first (leftmost) occurrence of substring `s2` and returns a pointer to the beginning of that substring. If the substring does not appear at all, `NULL` is returned. Unlike searching for a character, there is no library function that searches for the last occurrence of a substring. Figure 12.21 gives an example of using `strstr()`.

**Copying a string.** What does it mean to copy a string? Do we copy the pointer or the contents of the array? Both. We really need two copy operations because sometimes we want one effect, sometimes the other. Therefore, C has two kinds of operations: To copy the pointer, we use the assignment operator; to copy the contents of the array, we use `strcpy()` or `strncpy()`.

Figure 12.18 diagrams the result of using a pointer assignment; the expression `w6 = w5` copies the address from `w5` into `w6`. After the assignment, `w6` points at the same thing as `w5`. This technique is useful for output labeling. For example, in Figure 12.5 the value of `response` is set by assigning the address of the correct literal

---

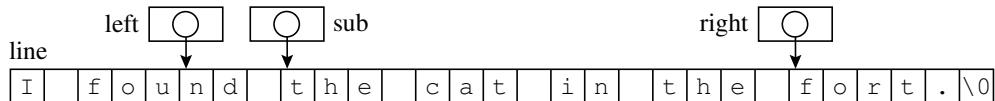
Walk down two strings at the same time. Leave loop if a pair of corresponding letters is unequal or if the end of one has been reached. Return a negative, zero, or positive result depending on whether the first string argument is less than, equal to, or greater than the second argument.

```
int my_strcmp( char* a, char* b )
{
    int k;
    for (k=0; a[k]; ++k){           // Leave loop on at end of string
        if (a[k] != b[k]) break;    // Leave loop on inequality
    }
    return a[k] - b[k];            // =0 if at end of both strings.
                                    // Negative if a is lexically before b.
                                    // Positive if b comes before a.
}
```

---

Figure 12.20. Possible implementation of `strcmp()`.

```
char line[] = "I found the cat in the fort.";
char * left, * right, * sub;
left = strchr( line, 'n' );
right = strrchr( line, 'f' );
sub = strstr( line, "the" );
```




---

Figure 12.21. Searching for a character or a substring in C.

```
char line[20];
strncpy( line, "Hotdog", 3 );    /* NO null terminator! */
strcpy( &line[3], "diggety" );
strcat( line, " dog!" );

line after strncpy()
H o t [REDACTED]

line after strcpy()
H o t d i g g e t y \0 [REDACTED]

line after strcat()
H o t d i g g e t y d o g ! \0 [REDACTED]
```

Figure 12.22. Search, copy and concatenate in C.

to it.

Copying the array portion of a string also is useful, especially when we need to extract data from an input buffer and move it into a more permanent storage area. The `strcpy()` function takes two string arguments, copies the contents of the second into the array referenced by the first, and puts a null terminator on the end of the copied string. It is necessary, first, to ensure that the receiving area is long enough to contain all the characters in the source string. This technique is illustrated in the second call of Figure 12.22. Here, the destination address is actually the middle of a character array.

The `strncpy()` function is similar to `strcpy()` but takes a third argument, an integer `n`, which indicates that copying should stop after `n` characters. Setting `n` equal to one less than the length of the receiving array guarantees that the copy operation will not overflow its bounds. Unfortunately, `strncpy()` does not automatically put a null terminator on the end of the copied string; that must be done as a separate operation unless the null character is copied by the operation. Or we can let a succeeding operation handle the termination, as happens in the example of Figure 12.22.

**String concatenation.** The function `strcat( string s1, string s2 )` appends `s2` onto the end of `s1`. Calling `strcat()` is equivalent to finding the end of a string, then doing a `strcpy()` operation starting at that address. If the position of the end of the string already is known, it actually is more efficient to use `strcpy()` rather than `strcat()` to append `s2`. In both cases, the array in which `s1` is stored must have enough space for the combined string. The `strcat()` function is used in the last call in the example of Figure 12.22 to complete the new string.

The function `strncat()` is like `strcat()`, except that the appending operation stops after at most `n` characters. A null terminator *is* added to the end of the string. (Therefore, up to  $n + 1$  characters may be written.) The careful programmer uses `strncpy()` or `strncat()`, rather than `strcpy()` or `strcat()`, and sets `n` to the number of storage slots remaining between the end of `s1` and the end of the array.

**Notes on Figure 12.23, C++ string operations.** All the string function used here are in the C++ string class so they are called using the name of a string object before the function name. In the code example, the object is named “line”, and it is used to call functions on lines 14–20.

- **find\_first\_of()**: This is like `strchr` in C, except that the return value is a subscript, not a pointer.
  - **find\_last\_of()**: This is like `strrchr` in C, except that the return value is a subscript, not a pointer.
  - **find()**: This finds a substring within a longer string. The subscript of the beginning of the leftmost occurrence is returned.
  - **replace( start, len, stringVar)**: The object written to the left of the function name will be modified. The first parameter is the subscript at which the replacement should `start`. The second parameter, `llen`, tells how many characters to replace from the old string. The third parameter is the new replacement string. (If it is longer than `len`, part will not be used.)

```

1 // -----
2 // Figure 12.23 Search, copy and concatenate in C++
3 // Rewritten in C++ June 21, 2016
4 // -----
5 #include "tools.hpp"
6
7 #define N 5
8
9 int main( void )
10 {
11     string line = "I found chili sauce";
12     int left, right, sub;
13
14     left = line.find_first_of( 'n' );    cout << left << " ";
15     right = line.find_last_of( 'f' );    cout << right << endl;
16     sub = line.find( "the" );           cout << sub << " This is string::npos. ";
17
18     line.replace( 0, 3, "Hotdog" );
19     line.replace( 3, 8, " diggety " );
20     line.append( " dog!" );
21     cout <<"\n" <<line <<"\n\n";
22 }
23
24 /* Output: -----
25 5 2
26 -1 This is string::npos.
27 Hot diggety chili sauce dog!
28 */

```

**Figure 12.23.** Search, copy and concatenate in C++.

- **append()**: The object written to the left of the function name will be modified by adding the new word on its end. The string will “grow” if needed. C
- **str::npos**: This is a constant defined in the string class, and used as an error return value. For example, if you search a string for the letter ‘x’, and it is not there, the return value will be **str::npos**. In my implementation, **str::npos** is defined as -1, however, that is not guaranteed. This value is returned from **find()** on line 16 and is displayed on line 21 of the output.

## 12.4 Arrays of Strings

An important aspect of C, and all modern languages, is that aggregate types such as strings and arrays can be compounded. Thus, we can build arrays of arrays and arrays of strings. An **array of strings**, sometimes called a **ragged array** because the ends of the strings generally do not line up in a neat column, is useful for storing a list of names or messages. This saves space over a list of uniform-sized arrays that are partially empty.

We can create a ragged array by declaring an array of strings and initializing it to a list of string literals.<sup>9</sup> For example, the following declaration creates the data structure shown in Figure 12.24:

```
char* word[3] = \{ "one", "two", "three" \};
```

This particular form of a ragged array does not permit the contents of the entries to be changed. Making a ragged array of arrays that can be updated is explained in a later chapter. The program in Figures 12.25 and ?? demonstrate the use of ragged arrays in two different contexts.

---

<sup>9</sup>In a later chapter, we show how to create a ragged array dynamically.

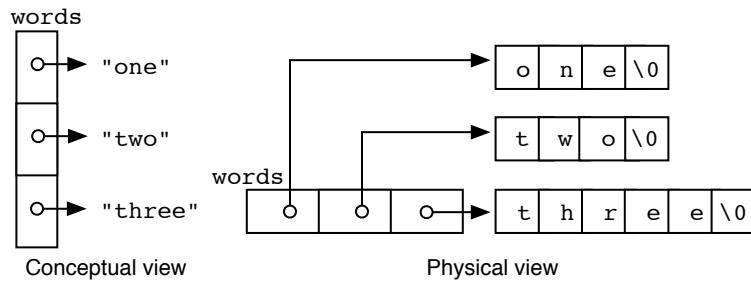


Figure 12.24. A ragged array.

#### 12.4.1 The Menu Data Structure

In computer programs, a menu is a list of choices presented interactively to a computer user. Each choice corresponds to some action or process that the program can carry out and the user might wish to select. In a modern windowing system, the user may make selections using a mouse. The other standard approach is to display an alphabetic or numeric code for each option and ask the user to enter the selected code from the keyboard. When execution of an option is complete, the menu is presented again for a new choice. Because they are easy to code and create an attractive interface, menus now are a component of programs in virtually every application area. Here, we present an easy way to set up your own menu-driven applications.

To use a menu, several things must occur:

- A title or some general instructions should be presented.
- Following that should be a list of possible choices and the input selection code for each. It is a good idea to include one option that means “take no action” or “quit.”
- Then the user must be prompted to enter a choice, and the choice must be read and validated.
- Finally, the program must carry out the user’s chosen intent.

Often, carrying out the chosen action means using supporting information specific to the choice. This information may be stored in a table implemented as a set of parallel arrays. In the menu display, each menu item can be shown next to its array subscript. The user is asked to choose a numbered item, and that number is used as a subscript for all of the arrays that form the table.

#### 12.4.2 An Example: Selling Ice Cream

This program will be given twice, first in C, then in C++ .The C program in Figure 12.25 illustrates the use of a menu of ice cream flavors and a parallel array of prices, as depicted at the top. A loop prints the menu. The loop variable is used as the subscript to print each menu item and is displayed as the selector for that item. When the user enters a choice, the number is used to subscript two **parallel arrays**: the **flavor** array and the **price** array. The program then prints a message using this information (and we imagine that ice cream and money change hands).

**Notes on Figure 12.25. Selecting ice cream from a menu.**

***First box: creating a menu.***

- The first string declared here is the greeting that will appear at the top of the menu.
- Second, we declare an array of strings named **flavor** and initialize it with the menu selections we want to offer.
- Another array, **price**, parallel to **flavor**, lists the price for a cone of each flavor. The position of each price in this array must be the same as the corresponding item in **flavor**.

This program builds the parallel-array data structure diagrammed below, then calls the `menu()` function in Figure 12.26.

flavor[0]	→	"Empty"	price[0]	0.00
flavor[1]	→	"Vanilla"	price[1]	1.00
flavor[2]	→	"Pistachio"	price[2]	1.50
flavor[3]	→	"Rocky Road"	price[3]	1.35
flavor[4]	→	"Fudge Chip"	price[4]	1.25
flavor[5]	→	"Lemon"	price[5]	1.20

```
#include <stdio.h>
#include <string.h>
#define CHOICES 6
int menu ( string title, int max, string menu_list[] );
int main( void )
{
    int choice;                                // The menu selection.
    string greeting = " I'm happy to serve you. Our specials today are: ";
    string flavor[CHOICES] = { "Empty", "Vanilla", "Pistachio",
                               "Rocky Road", "Fudge Chip", "Lemon" };
    float price[CHOICES] = { 0.00, 1.00, 1.50, 1.35, 1.25, 1.20 };

    choice = menu( greeting, CHOICES, flavor );
    printf( "\n Here is your %s cone.\n That will be $%.2f. ",
            flavor[choice], price[choice] );
    puts( "Thank you, come again." );
}
```

Figure 12.25. Selecting ice cream from a menu.

- Making the menu longer or shorter is fairly easy: Just change the value of `CHOICES` at the top of the program, add or delete a string to or from the `flavor` array, and add or delete a corresponding cost to or from the `price` array.
- Since this is a relatively small program, the declarations have been placed in `main()`. Alternative placements are discussed as they occur in other examples in this chapter.

*Second box: calling the menu() function.*

- Since the `menu()` function, defined in Figure 12.26, will be used for all sorts of menus in various programs, the program must supply, as an argument, an appropriate application title to be displayed above the menu items.
- The second argument is the number of items in the menu. In C, the number of items in an array must be an argument to any function that processes the array; the function cannot determine this quantity from just the array itself.
- The third argument is the name of the array that contains the menu descriptions. These descriptions will be displayed for the user next to the selection codes.
- The return value from this function is the index of the menu item that the user selected. The `menu()` function returns only valid selections, so we can safely store the return value and use it as a subscript with no further checking.

*Third box: the output.*

- The variable `choice` is used to subscript both arrays. The program prints the chosen flavor and the price for that flavor using the validated menu selection as a subscript for the `flavor` and `price` arrays, respectively.
- Below the menu presented on the screen (shown later), the user will see his or her selection, as well as the results, displayed like this:

---

This function is called from the main program in Figure 12.25.

```

int menu( char* title, int max, char* menu_list[] ) {
    int choice;
    printf( "\n %s\n ", title );
    for (int n = 0; n < max; ++n)  printf( "\t %i. %s \n", n, menu_list[n] );

    printf( " Please enter your selection: ");
    for(;;) {                      // Prompt for and validate a menu selection.
        scanf( "%i", &choice );
        if (choice >= 0 && choice < max) break; // Accept valid choice.
        printf( " Please enter number between 0 and %i: ", max - 1 );
    }

    return choice;
}
```

---

**Figure 12.26. A menu-handling function.**

```

Please enter your selection: 3
Here is your Rocky Road cone.
That will be $1.35. Thank you, come again.
```

#### Notes on Figures 12.26, 12.27, and 12.28. A menu-handling function.

##### *First box: the function header.*

- The first parameter is a string that contains a title for the menu. This should be declared with the menu array, as in Figure 12.25.
- The second parameter is the number of choices in the menu. This number is used to control the loop that displays the menu and determine which index inputs are legal and which are out of bounds.
- The final parameter is an array of strings that contains the list of menu items.
- The return value will be a legal subscript for the arrays that contain the menu and price information.

##### *Second box: displaying the menu.*

- First, the program prints the title for the menu with appropriate vertical spacing.
- Then it uses a loop to display the menu. On each repetition, the loop displays the loop counter and one string. This creates a numbered list of items, where each number displayed is the array subscript of the corresponding item.
- The actual menu display follows. Note that choice 0 permits an escape from this menu without buying anything. It is a good idea to include such a “no operation” alternative. A “quit” option is not necessary in this menu since it is displayed only once by the program.

```

I'm happy to serve you. Our specials today are:
0. Empty
1. Vanilla
2. Pistachio
3. Rocky Road
4. Fudge Chip
5. Lemon
```

```

1 // -----
2 // Figure 12.27: Selecting ice cream from a menu in C++.
3 // Rewritten in C++ June 18, 2016
4 // -----
5 #include "tools.hpp"
6
7 #define CHOICES 6
8 int menu ( const char* title, int max, const char* menuList[] );
9
10 int main( void )
11 {
12     int choice;                                // The menu selection
13     const char* greeting = "\n I'm happy to serve you. Our specials today are: ";
14     const char* flavor[CHOICES] = { "Empty", "Vanilla", "Pistachio",
15                                     "Rocky Road", "Fudge Chip", "Lemon" };
16     float price[CHOICES] = { 0.00, 1.00, 1.50, 1.35, 1.25, 1.20 };
17
18     choice = menu( greeting, CHOICES, flavor );
19     cout << "\n Here is your " << flavor[choice] <<" cone. That will be $" 
20           <<fixed <<setprecision(2) << price[choice] <<"\n Thank you, come again. \n";
21 }
22
23 // -----
24 // Display a menu then read and validate a numeric menu choice.
25 int
26 menu ( const char* title, int max, const char* menuList[] )
27 {
28     int choice;                                // To store the menu selection.
29     printf( "\n %s\n", title );
30     for (int n=0; n < max; ++n)    cout << "\t " <<n << ". " <<menuList[n] <<endl;
31
32     cout << " Please enter your selection: ";
33     for(;;) {                                  // Prompt for and validate a menu selection.
34         cin >> choice;
35         if (choice >= 0 && choice < max) break;    // Accept valid choice.
36         cout << " Please enter a number between 0 and " << max-1 <<endl;
37     }
38     return choice;
39 }
40 /* Output: -----
41 I'm happy to serve you. Our specials today are:
42     0. Empty
43     1. Vanilla
44     2. Pistachio
45     3. Rocky Road
46     4. Fudge Chip
47     5. Lemon
48 Please enter your selection: 2
49
50 Here is your Pistachio cone. That will be $1.50
51 Thank you, come again.
52 -----
53 I'm happy to serve you. Our specials today are:
54     0. Empty
55     1. Vanilla
56     2. Pistachio
57     3. Rocky Road
58     4. Fudge Chip
59     5. Lemon
60 Please enter your selection: 6
61 Please enter a number between 0 and 5
62 ----- */

```

Figure 12.27. Buying a Cone in C++.

**Third box: the prompt, input, and menu selection validation.**

- The original prompt is written outside the loop, since it will be displayed only once. If the first selection is invalid, the user will see an error prompt.
  - This **for** loop is a typical data validation loop. It is very important to validate an input value before using it as a subscript. If that value is outside the range of legal subscripts, using it could cause the program to crash. The smallest legal subscript always is 0. The second argument to this function is the number of items in the menu array, which is one greater than the maximum legal subscript. The program compares the user's input to these bounds: If the selection is too big or too small, it displays an error prompt and asks for a new selection.
  - A good user interface informs the user of the valid limits for a choice after he or she makes a mistake. Otherwise, the user might not understand what is wrong and have to figure it out by trial and error.
- 

```

1 // -----
2 // Figure 12.28: Read a menu choice and use it without validation.
3 // Rewritten in C++ June 18, 2016
4 // -----
5 int
6 menu ( const char* title, int max, const char* menuList[] )
7 {
8     int choice;           // To store the menu selection.
9     printf( "\n %s\n", title );
10    for (int n=0; n < max; ++n)    cout << "\t " <<n << ". " <<menuList[n] <<endl;
11
12    cout << " Please enter your selection: ";
13    cin >> choice;
14    return choice;
15 }
16
17 /* Output: -----
18 I'm happy to serve you. Our specials today are:
19     0. Empty
20     1. Vanilla
21     2. Pistachio
22     3. Rocky Road
23     4. Fudge Chip
24     5. Lemon
25 Please enter your selection: -1
26
27 Here is your ?? cone. That will be $0.00
28 Thank you, come again.
29 -----
30
31 I'm happy to serve you. Our specials today are:
32     0. Empty
33     1. Vanilla
34     2. Pistachio
35     3. Rocky Road
36     4. Fudge Chip
37     5. Lemon
38 Please enter your selection: 6
39
40 Segmentation fault
41 ----- */

```

---

Figure 12.28. Using an invalid subscript.

- When control leaves the loop, `choice` contains a number between 0 and `max-1`, which is a legal subscript for a menu with `max` slots. The function returns this choice. The calling program prints the chosen flavor and the price for that flavor using this number as a subscript for the `flavor` and `price` arrays.
- The following output demonstrates the validation process. The `menu()` function does not return to `main()` until the user makes a valid selection. After returning, `main()` uses the selection to print the chosen flavor and its price:

```
I'm happy to serve you. Our specials today are:
 0. Empty
 1. Vanilla
 2. Pistachio
 3. Rocky Road
 4. Fudge Chip
 5. Lemon
Please enter your selection: -3
Please enter number between 0 and 5: 6
Please enter number between 0 and 5: 5

Here is your Lemon cone.
That will be $1.20. Thank you, come again.
```

- The `menu()` function in Figure 12.26 must validate the input because C and C++ do not guard against the use of meaningless subscripts. To demonstrate the effects of using an invalid subscript, we removed the data validation loop from the third box, leaving only the prompt and the input. See Figure refex-badsub-func.

As you can see, C++ calculates a machine address based on any subscript given it and uses that address, even though it is not within the array bounds. The results normally are garbage and should be different for each illegal input. Sometimes the output is simply garbage, as in the first example. On some computer systems, a segmentation error will probably cause the program to crash and may force the user to reboot the computer. A well-engineered program does not let an input error cause a system crash or garbage output. It validates the user's response to ensure that it is a legal choice, as in Figure 12.26.

## 12.5 String Processing Applications

In this section, we examine two applications that use many of the string processing functions presented in the chapter.

### 12.5.1 Password Validation

The following program uses some of the string functions in a practical context; namely, requiring the user to enter a password to access a sensitive database on the computer. This code could be part of many applications and will be used in Chapter 14 in a complete program.

Figure 12.29 is a main program that performs password validation, then calls an application function. In this example, the password is a constant in the main program, and it is not encrypted. In a real application, it would be encrypted and it would not be constant.

#### Notes on Figures 12.29 and 12.30. Checking a password.

##### *First box : A limit on string length.*

- `BUFLLEN` is used to define the variable `word`, which will store the user's password input, and it is used in the call on `scanf()` to read the password from the user. This limits passwords to 79 characters plus a null terminator, an arbitrary length that is much longer than we expect to need. 80 characters is a common limit for the length of an input line. This usage can be traced back to the time when punched cards were used for computer input. A punched card had 80 columns.
- In C++, no length limit is needed because a `string` will be allocated that is big enough to hold anything that is entered.

---

Create a personalized form letter to send to each members of the class.

```
#include <stdio.h>
#include <string.h> // for strlen(), strcmp(), strcpy(), and strchr()
#include <ctype.h> // for isspace()

#define BUflen 80
void doTask( void ){ puts( "Entering doTask function\n" ); }

// -----
int main( void )
{
    char password[] = "StaySafe";
    char word[ BUflen ];

    puts( "\nABC Corporation Administrative Database" );
    printf( "\nPlease enter the password: " );
    scanf( "%79[^\\n]", word );

    if (strcmp( word, password )) printf( "No Entry! \n" );
    else doTask(); // Application logic would be in this function.

    return 0;
}
```

---

**Figure 12.29. Password validation: Comparing strings.**

**Second box and C++ lines 11–12: Defining the password.**

- In a real application, the owner of the application would be able to change the password. However, we are trying to keep this program simple and so we are using a literal constant as the password. The program would need to be recompiled to change it.
- In C, character arrays are used for both the password and the input variable. The password array is 9 characters long: enough to store the quoted string and a null.
- In C++, type string is used for both the password and the input variable. We could use character arrays, just as we do in C, but strings are preferred because they are less prone to error. We do not need to worry about limiting the length of a string.

**Third box and C++ lines 14–16: Entering the password.**

- We prompt the user to enter a password. In a real application, this would be done without showing the password on the screen. In this program, however, we let the password stay on the screen for simplicity, and because we would need to go outside standard C to mask the password input.
- This call on `scanf()` will read all the characters typed, up to but not including the end-of-line character, and store them in `word`. A maximum of 79 characters will be read, to avoid overflowing the buffer. If more characters are typed, the extra ones will remain, unread, in the input stream.
- In C++, we want to read into a string, not a char array so we do not use `cin >>`. The form of `getline` with two parameters reads from an input stream into a string.

**Fourth box and C++ lines 18–19: the comparison.**

- `strcmp()` takes two C string arguments of any variety. Here, the arguments are both character arrays, but one could be a literal string.
- If the two strings are unequal, `strcmp()` will return a non-zero value and an error comment will be printed.
- If the two strings are equal, the `else` clause will be selected. The user now has gained entry into the protected part of the program, which calls the `doTask()` function.

- In C++, it is possible to use `==` to compare the content of two C++ strings. This is much less error-prone than the old C version. This is a good reason to use C++ strings in your programs instead of C strings.
- The C++ program shows output from successful and unsuccessful login attempts.

### 12.5.2 The `menu_c()` Function

The process of displaying the menu and eliciting a selection is much the same for any menu-based program: the content of the menu changes from application to application but the processing method remains the same. There is one major variation: some menus are character-based and the menu choice is type `char`, (not type `int`) and is processed by a `switch` (not by using subscript). The commonality from application to application enables us to write two general-purpose menu-handling functions: `menu()`, for integer choices, is shown above in Figure 12.26, and `menu_c()`, for character choices, is in Figure 12.31.

Like the `menu()` function, `menu_c()` receives a menu title, the number of menu items, and the menu array as arguments. Now, however, each menu string now consists of a selection code character and a phrase describing the menu item. The fourth argument is a string consisting of all the letters that are legal menu choices, and is used to validate the selection.

The function displays the strings from the menu array, one per line, then reads and returns the menu choice

```

1 // -----
2 // Figure 12.29: Checking a password in C++.
3 // Rewritten in C++ June 24, 2016
4 //
5 #include "tools.hpp"
6 void doTask( void ){ cout <<"Entering doTask function" <<endl; }
7 //
8 //
9 int main( void )
10 {
11     string password = "StaySafe";
12     string word;
13
14     cout<<( "\nABC Corporation Administrative Database"
15             "\nPlease enter the password: " );
16     getline( cin, word );
17
18     if ( word == password ) doTask();
19     else cout <<"No Entry!" <<endl;
20
21     return 0;
22 }
23
24 /* Output: -----
25
26 ABC Corporation Administrative Database
27 Please enter the password: StaySave
28 No Entry!
29 -----
30
31 ABC Corporation Administrative Database
32 Please enter the password: StaySafe
33 Entering doTask function
34 */

```

Figure 12.30. Password validation: Comparing strings.

---

This function displays a menu, then reads, validates, and returns a non-whitespace character. It is called from Figures 12.34.

```

char menu_c( string title, int n, const string menu[], string valid )
{
    char ch;
    printf("\n%s\n\n", title);
    for(;;) {
        for( int k=0; k<n; ++k ) printf("\t%s \n", menu[k]);
        printf("\n Enter code of desired item: ");
        scanf(" %c", &ch);
        if (strchr( valid, ch )) break;
    }
    while (getchar() != '\n');      // Discard rest of input line
    puts("Illegal choice or input error; try again.");
}
return ch;
}

```

---

**Figure 12.31.** The `menu_c()` function.

in the form of a single nonwhitespace character.<sup>10</sup> This function will be used in the next program, Figure 12.34.

**Notes on Figures 12.31 and 12.32. The `menu_c()` and `doMenu()` functions.** We expect the third parameter (the menu) to have a phrase describing each choice and, with that phrase, a letter to type. This function displays a title and a menu and prompts for a response in the same manner as `menu()`, in Figure 12.26. After validation, the response is returned.

*First box: Reading and validating the choice.*

---

<sup>10</sup>In contrast, `menu()` reads and returns an integer response that can be used directly as a table subscript.

---

```

1 // -----
2 // Figure 12.32: Read and validate an alphabetic menu choice character.
3 // -----
4 char
5 doMenu( cstring title, int n, cstring menItems[], string valid )
6 {
7     char ch;
8     cout <<"\n" <<title <<"\n\n";
9     for(;;) {
10         for( int k=0; k<n; ++k ) cout <<"\t" <<menItems[k] <<"\n";
11         cout <<"\n Enter code of desired item: ";
12         cin >>ch;
13         if ( valid.find_first_of(ch) != string::npos ) break;
14         cin.ignore(255);           // Skip inputs up to first newline.
15         cout <<"Illegal choice or input error; try again.\n";
16     }
17     return ch;
18 }

```

---

**Figure 12.32.** `menu_c` in C++.

- After the for loop displays the menu, the `printf()` prompts the user to make a selection. It lets the user know that a character (not a number) is expected this time. This is almost the same as lines l11 and 12 in the C++ version
- We use `scanf()` with " %c" to read the user's response. The space before the %c is important because the input stream probably contains a newline character from some prior input step.<sup>11</sup> The program must skip over that character to reach the user's current input.

Line 13 of the C++ version does this task. It is much simpler because `>>` always skips leading whitespace.

- To validate the user's choice, we compare it to the list of valid choices that were supplied as the fourth parameter. The `strchr()` function makes this very easy: if the user's choice is in the list, `strchr()` will return a pointer to that letter. If it is not in the list, `strchr()` will return a NULL pointer. Thus, a non-null return value indicates a valid choice. We use the `if` statement to test this value. If the choice is valid, control leaves the loop. Otherwise, we move on to the error handling code.

In C++, the `find_first_of()` function makes the same search, but it returns a subscript, not a pointer. If the input char is not in the list of legal inputs, the function returns `str::npos`. Any other return value indicates a valid input choice.

#### *Second box: Error handling.*

- This is a one-line loop with no loop body; the semicolon is not a mistake. It reads a character and tests it, reads and tests, until the newline is found. The purpose is to skip over any extra characters that the user might have typed and that might be still in the input stream. Cleaning out the input stream is often done after an error to help the user become synchronized with the action of the program and what should happen next.

Line 15 of the C++ version does this job using a built-in stream manipulator, `ignore`. This line will remove up to 255 chars from the input stream, but stops when a newline is found.

- We announce clearly that an error was made. Control will return to the top of the loop and display the menu again. The only way to leave the loop is to enter a valid choice. This is done on line 16 of the C++ version.

### 12.5.3 Menu Processing and String Parsing

The `switch` statement was introduced back in Section 6.3 as a method of decision making. Putting a `switch` inside a loop is an important control pattern used in menu processing, especially when the menu choices are characters rather than numbers. The loop is used to display the menu repeatedly, until the user wishes to quit, while the `switch` is used to process each selection. One option on the menu should be to quit, that is, to leave the loop.

This example program starts with a flow diagram of the main function, followed by the main function in C, the main function in C++, and notes on these three Figures. Following that are the subfunctions in both languages and notes on the subfunctions.

**Notes on Figures 12.33, 12.34 and 12.35. Magic memo maker and its flow diagram.** This flow chart is the same for both C and C++ versions.

#### *First box: the menu.*

- This box corresponds to lines 8–12 in the C++ version.
- As usual, a menu consists of an integer, `N`, and a list of `N` strings to be displayed for the user. This menu is designed to be used with `menu_c()`, so each string incorporates both a single-letter code and a brief description of the option.
- To add another menu option, we must increase `N`, add a string to the array, and add another case in the switch statement (second box). To make such modifications easier, the declarations have been placed at the top of the program. The `const` qualifier is used to protect them from accidental destruction, since they are global.

---

<sup>11</sup>Review the discussion of these issues concerning problems with `getchar()` following Figure 8.6.

**Second box: the menu loop.**

- This box corresponds to lines 23–45 in the C++ version.
- This menu-processing loop is typical and can be adapted to a wide variety of applications. Its basic actions are

An example of this control structure is given in Figures 12.34 and 12.33. The program in Figure 12.34 presents a menu and processes selections until the user types the code q or Q. It illustrates the use of a **for** loop with a **switch** statement inside it to process the selections, a **continue** statement to handle errors, and a **break** statement to quit. This program also shows a use of **strchr()** and **strrchr()** in context.

- The first inner box (lines 24–26 in C++) reads the user's choice. We use **menu\_c()** or **coMenu()** to display a menu and return a validated selection. Then we convert the selection to upper case and break out of the **for** loop if the user has chosen the "Quit" option.
- The second inner box (lines 28–40 in C++) receives the choice and we use a **switch** to process the three legal action codes. A case is defined for each one that sets two variables that will be used later to print parts of the message. Figure 12.36 shows how we set the value of the string variable **re** for the memo generator: The **break** at the end of each valid case takes control to the processing statements at the bottom of the loop.
- We do not need a default case in this program because the **menu\_c()** function validates the input and

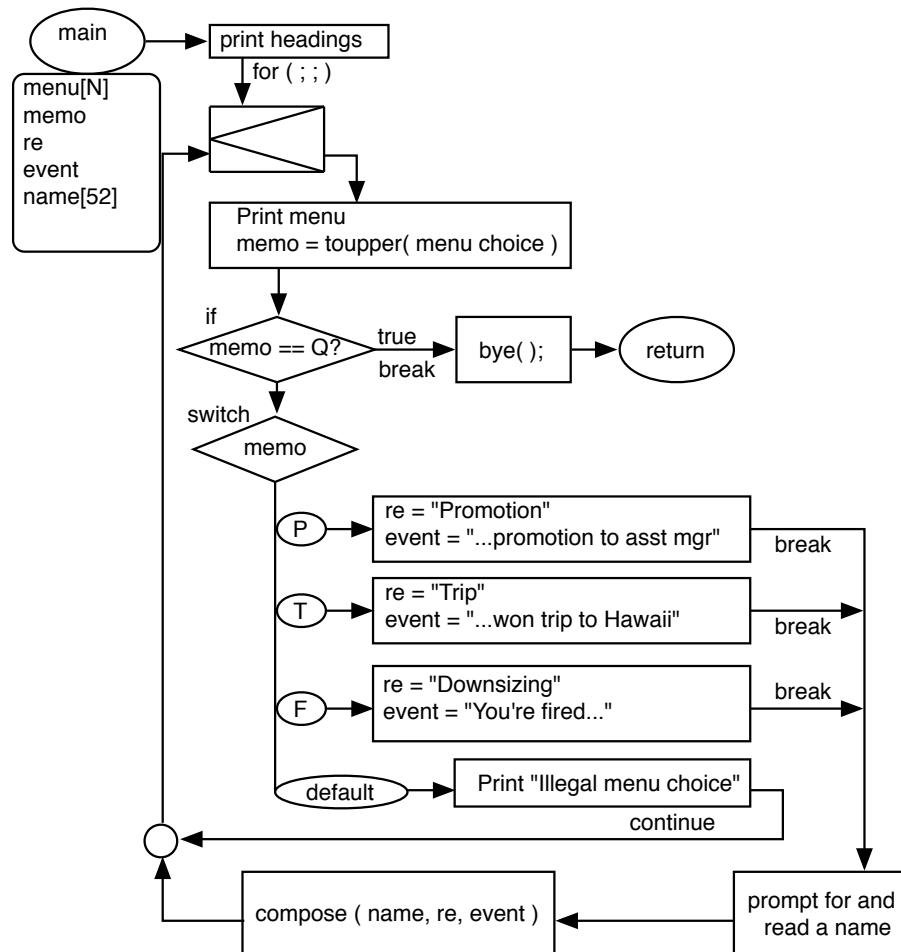


Figure 12.33. Flow diagram for Figure 12.34.

This program calls the `compose()` function from Figure 12.37. Its control flow is diagrammed in Figure 12.33.

```
#include <stdio.h>
#include <string.h> // For strrchr().
#include <ctype.h> // For toupper().

void compose( char* name, char* re, char* event ); // Modifies name.

#define N 4           // Number of memo menu choices
const char* menu[N] = {"P Promote", "T Trip", "F Fire", "Q Done"};
```

```
int main( void )
{
    char memo;           // Menu choice.
    char *re, *event;   // Subject and main text of memo.
    char name[52];      // Employee's complete name.

    puts( "\n Magic Memo Maker" );

    for(;;) {
        memo = toupper( menu_c( " Next memo:", N, menu ) );
        if (memo == 'Q') break; // Leave for loop and end program.

        switch (memo) {
            case 'P': re = "Promotion";
                        event = "You are being promoted to assistant manager.";
                        break;
            case 'T': re = "Trip";
                        event = "You are tops this year "
                                "and have won a trip to Hawaii.";
                        break;
            case 'F': re = "Downsizing";
                        event = "You're fired.\n"
                                "Pick up your final paycheck from personnel.";
                        break;
        }

        printf( " Enter name: " );
        scanf( " %51[^\\n]", name );
        compose( name, re, event );
    }

    return 0;
}
```

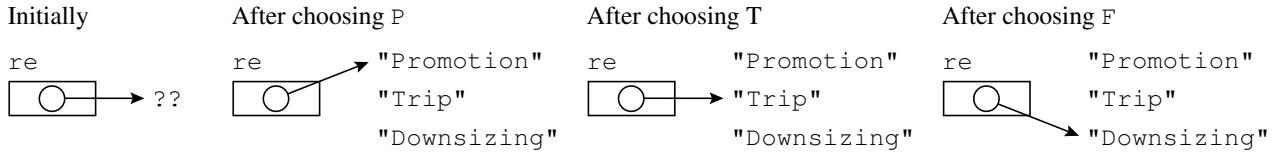
Figure 12.34. Magic memo maker in C.

```

1 // -----
2 // Figure 12.32: Magic memo maker in C++.
3 // Rewritten in C++ June 21, 2016
4 // -----
5 #include "tools.hpp"
6 typedef const char* cstring;
7
8 #define N 4           // Number of memo menu choices
9 #define BOSS "Leland Power"
10 cstring menItems[] = {"P Promote", "T Trip", "F Fire", "Q Done"};
11 const string valid( "PpTtFfQq" ); // The valid menu choices for this app.
12
13 char doMenu( cstring title, int n, cstring menItems[], string valid );
14 void compose( string name, cstring re, cstring event );
15
16 int main( void )
17 {
18     char memo;           // Menu choice.
19     cstring re, event;   // Subject and main text of memo.
20     string name;         // Employee's complete name.
21
22     cout <<"\n Magic Memo Maker\n";
23     for(;;) {
24         char ch = doMenu( " Next memo:", N, menItems, valid );
25         memo = toupper( ch );
26         if (memo == 'Q') break; // Leave for loop and end program.
27
28         switch (memo) {
29             case 'P': re = "Promotion";
30                         event = "You are being promoted to assistant manager.";
31                         break;
32             case 'T': re = "Trip";
33                         event = "You are tops this year "
34                             "and have won a trip to Hawaii.";
35                         break;
36             case 'F': re = "Downsizing";
37                         event = "You're fired. \n"
38                             "Pick up your final paycheck from personnel.";
39                         break;
40         }
41         cout <<" Enter name: ";
42         cin >> ws;
43         getline( cin, name );
44         compose( name, re, event );
45     }
46     return 0;
47 }
```

---

Figure 12.35. Magic memo maker in C++.



**Figure 12.36.** Using a string variable with a menu.

supplies it for the switch.

- The request is processed in the third inner box (lines 41–44 in C++). The common actions for generating the memo are performed after the end of the `switch`. When control reaches this point, invalid menu choices have been eliminated and valid ones fully set up. In this program, we call the `compose()` function to process the request.

The output from the main program begins like this:

```
Magic Memo Maker
Next memo:

P Promote
T Trip
F Fire
Q Done

Enter letter of desired item: T
Enter name: Harvey P. Rabbit, Jr.
```

In Figure 12.37, the memo is composed and processed.

#### Notes on Figure 12.37. Composing and printing the memo.

##### *First box: memo header.*

- This box corresponds to C++ lines 57–58.
- The program prints a memo header that contains the employee's entire name, the boss's name, and the subject of the memo.
- The boss's name is supplied by a `#define` command. The other information is passed as arguments from `main()`.

##### *Second box: finding the end of the last name.*

- This box corresponds to the C++ version, lines 60 and 61. The string parameter is modified in the C version but in C++, parts of it are copied to other variables instead.
- A name can have up to four parts: first name, middle initial (optional), last name, and a comma followed by a title (optional).
- For the salutation, we wish to print only the employee's last name, omitting the first name, middle initial, and any titles such as Jr. or V.P. Since the initial and titles are optional, it is a nontrivial task to *locate* the last name and separate it from the rest of the name. The last name is followed by either a comma or the null terminator that ends the string.
- In C++, we search the name for a comma thus:

```
extra = name.find_first_of( ',', );           // Find end of the last name.
if (extra != string::npos) lname = name.substr(0,extra);
```

If the result is `str::npos`, we know that the last name is the last thing in the string. We then make a copy of the head of the string, up to the comma at the end of the last name. This isolates the name and eliminates the extra part.

- In C, we search the name for a comma thus:

This function is called from Figure 12.34. As a side effect, the first argument, `name`, is modified.

```
#define BOSS "Leland Power"

void compose( char* name, char* re, char* event )
{
    char* extra, * last;           // Pointers used for parsing name.
    printf( "\n\n To: %s\n", name );
    printf( " From: The Big Boss\n" );
    printf( " Re: %s\n\n", re );

    extra = strchr( name, ',' );      // Find end of last name.
    if (extra != NULL) *extra = '\0'; // Mark end of last name.

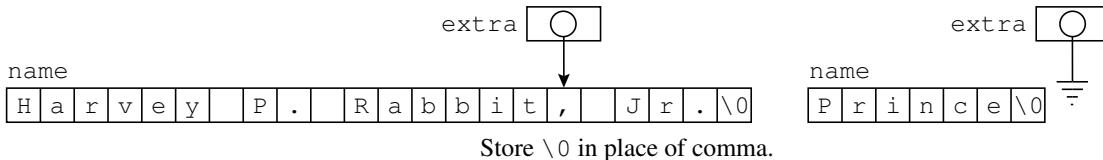
    last = strrchr( name, ' ' );     // Find beginning of last name.
    if (last == NULL) last = name;
    else ++last;

    printf( " Dear M. %s:\n %s\n\n -- %s\n\n", last, event, BOSS );
}
```

Figure 12.37. Composing and printing the memo in C.

```
extra = strchr( name, ',' );
if (extra != NULL) *extra = '\0';
```

If the result is `NULL`, we know that the last name is the last thing in the string. Otherwise, the result of `strchr()` is a pointer to the comma. The following diagram shows how the `extra` pointer would be positioned for two sample inputs, with and without “extra” material at the end:



We replace the comma, if it exists, with a null character, using `*extra = '\0'`. We now know for certain that the last name is followed by a null character. This has the side effect of changing the value of the first argument of `compose()` by shortening the string. The part cut off is colored gray in the next diagram. This is an efficient but generally risky technique. Documentation must be provided to warn programmers not to call the function with constants, literal strings, or strings that will be used again. For a safe alternative, use the C++ version, where the nMW is copied into a local variable and the original remains unchanged.

#### *Third box: the beginning of the last name.*

- This corresponds to lines 63 and 64 of the C++ version.
- To find the beginning of the last name in the (newly shortened) string using C++, call `lname.find_last_of(' ')`. The result is the subscript of the rightmost space in the lname. If a space is found, the last name will be between that space and the end of the string. The program then uses the `substr()` function and assignment to copy the last name into lname. You can safely assign one C++ string to another.

If the name has only one part, like Midori or Prince, the result stored in `last`, will be `str::npos`. In that case, the last name is the first and only thing in the string and is ready to use.

- To find the beginning of the last name in C, use `strrchr(name, ' ')` to search the (newly shortened) string for the rightmost space. The result is indicated by a dashed arrow in the diagram that follows.

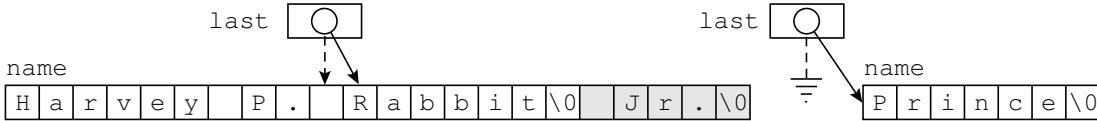
```
48 // -----
49 // Figure 12.35: Composing and printing the memo in C++.
50 // -----
51 void
52 compose( string name, cstring re, cstring event )
53 {
54     int extra, last;                      // Subscripts used for parsing name.
55     string lname = name;
56
57     cout <<"\n\n To: " <<name <<"\n From: The Big Boss\n";
58     cout <<" Re: " <<re <<"\n\n";
59
60     extra = name.find_first_of( ',' );        // Find end of the last name.
61     if (extra != string::npos) lname = name.substr(0,extra); // eliminate extra.
62
63     last = lname.find_last_of( ',' );         // Find beginning of the last name.
64     if (last != string::npos) lname = lname.substr(last+1);
65
66     cout << " Dear M. " <<lname <<":\n" <<event <<"\n\n -- " <<BOSS <<"\n\n";
67 }
68
69
70
71 /* Output: -----
72
73 Magic Memo Maker
74
75 Next memo:
76
77 P Promote
78 T Trip
79 F Fire
80 Q Done
81
82 Enter code of desired item: p
83 Enter name: Anden
84
85
86 To: Anden
87 From: The Big Boss
88 Re: Promotion
89
90 Dear M. Anden:
91 You are being promoted to assistant manager.
92
93 -- Leland Power
94 ----- */
```

---

Figure 12.38. Composing and printing the memo in C++.

---

```
last = strrchr( name, ' ' );
if (last == NULL) last = name;
else ++ last;
```



- If a space is found, as in the diagram on the left, the last name will be between that space and the (newly written) null terminator. The program then increments the pointer one slot so that it points at the first letter of the name.
- If the name has only one part, like Midori or Prince, the result of `strrchr()`, which is stored in `last`, will be `NULL`. In that case, the last name is the first and only thing in the string, so we set `last = name`.

**Fourth box and line 66: printing the memo.** The program now has located the last name and is ready to print the rest of the memo. The event (an input parameter) and the boss's name (given by a `#define` command) are used. Here is a sample output memo:

```
To: Harvey P. Rabbit, Jr.
From: The Big Boss
Re: Trip

Dear M. Rabbit:
You are tops this year and have won a trip to Hawaii.

-- Leland Power
```

## 12.6 What You Should Remember

### 12.6.1 Major Concepts

- The type name `string` is not a built-in name in C; use a `char` array when you want to store a string, and use a `char*` to point at a string in an array.
- The type name `string` IS standard in C++. The old C types also work in C++ and must be used for a variety of purposes.
- It is not possible to change the contents of a string literal.
- C's standard `string` library contains many useful functions for operating on strings. In this chapter, we note the following:
  - To find the number of characters in a string, use `strlen()`. (Note that `sizeof` gives the size of the pointer part.)
  - To search a string for a given character or substring, use `strchr()`, `strrchr()`, and `strstr()`.
  - To compare two strings for alphabetic order, use `strcmp()` and `strncmp()`. (Note that `==` tells only whether two string pointers point at the same slot in an array.)
  - To copy a string or a part of a string, use `strcpy()`, `strncpy()`, `strcat()`, and `strncat()`. (Note that `=` copies only the pointer part of the string and cannot be used to copy an array.)
- C++'s standard `string` library a similar collection of useful functions for operating on C++ strings. In this chapter, we have used the following:
  - To find the number of characters in a string, use `tlength()`.
  - To search a string for a given character or substring, use `find_first_of()`, `find_last_of()`, and `find()`.
  - To compare two strings for alphabetic order in C++, use `compare()`.
  - To copy a string into a string variable, use `=`. To copy a part of a string, use `replace()` or `append()`.

- A C string or a character array may be initialized with a string literal. The length of the array must be at least one longer than the number of letters in the string to allow for the null terminator.
- A C++ string may be initialized with any string literal of any length.
- Literal strings, `char*` variables, and character arrays all are called *cstrings*. However, these objects have very different properties and are not fully interchangeable.
- In memory, a cstring is represented as a pointer to a null-terminated array of characters. You can point at any character in the array.
- A C++ string is a compound data object with two integer parts and a cstring part.
- A string variable is a pointer and must be set to point at some referent before it can be used. The referent can be either a quoted string or a character array.
- The operators that work on the pointer part of a C string are different from the functions that do analogous jobs on the array part. Both are different from the operators that work with C++ strings.
- An array of strings, or ragged array, is a compound data structure that often saves memory space over an array of fixed-length strings. One common use is to represent a menu.
- An essential part of interactive programs is the display of a menu to provide choices to the user. It is essential to validate menu choices before using them.

A `switch` statement embedded in a loop is a common control structure for processing menus. The menu display and choice selection continue until the user picks a termination option.

### 12.6.2 Programming Style

- A loop that processes all of the characters in a string typically is a sentinel loop that stops at the null terminator. Many string functions operate in this manner.
- Use the “zero” literal of the correct type. Use `NULL` for pointers in C, `\0` for characters, and `""` for strings. Reserve `0` for use as an integer. Use `nullptr` for pointers in C++.
- You can use a string literal to initialize a character array or a string variable. For the array, this is preferable to a sequence of individually quoted characters.
- Adjacent string literals will be merged. This is helpful in breaking up long output formats in a `printf()` statement.
- Use `char[]` as a parameter type when the argument is a character array that may be modified. Use `char*` when the argument is a literal or a string that should not be modified.
- Declare a `char` array to hold text input or for character manipulation and take precautions that the operations will not go past the last array slot. Do not attempt to store input in a `char*` variable.
- Using the brackets specifier makes `scanf()` quite versatile, allowing you to control the number of characters read as well as the character or characters that will terminate the input operation.
- A menu-driven program provides a convenient, clear user interface. To use a menu, a list of options must be displayed where each option is associated with a code. The user is instructed to select and key in a code, which then is used to control the program’s actions. Having one option on the menu to either quit or do nothing is common practice. User selections should be validated.
- One or more data arrays parallel to a menu array can be used to make calculations based on a table of data. Each array in the set represents one column in the table that relates to one data property. If integer selections are used, the items in the arrays can be indexed directly using the choice value.
- The process of using a menu is much the same for all menu applications. We introduced two menu functions that automate the process, `menu()` for numeric codes and `menu_c()` for alphabetic codes. The ragged arrays used for menus should be declared as `const` if they are defined globally.

### 12.6.3 Sticky Points and Common Errors

These errors are based on a misunderstanding of the nature of strings.

**Ampersand errors.** When using `scanf()` with `%s` to read a string, the argument must be an array of characters. Do *not* use an ampersand with the array name, because all arrays are passed by address. A character pointer (a string) can be used as an argument to `scanf()`, but it first must be initialized to point at a character array. Do *not* use an ampersand with this pointer because a pointer already is an address, and `scanf()` does not want the address of an address.

**String input overflow.** When reading a string as input, limit the length of the input to one less than the number of bytes available in the array that will store the data. (One byte is needed for the null character that terminates the string.) C provides some input operations, such as `gets()`, that should not be used because there is no way to limit the amount of data read. If an input operation overflows the storage, it will destroy other data or crash the program.

**No room for the null character.** When you allocate an array to store string data, be sure to leave space for the null terminator. Remember that most of the library functions that read and copy strings store a null character at the end of the data.

**A pointer cannot store character data.** Be careful to distinguish between a character array and a variable of type `char*`. The first can store a whole sequence of characters; the second cannot. A `char*` variable can only be used to point at a literal or a character array that has already been created.

**Subscript validation.** If the choice for a menu is not validated and if it is used to index an array, then an erroneous choice will result in accessing a location outside of the array bounds. This may cause almost any kind of error, from garbage in the output to an immediate crash.

**The correct string operator.** Strings in memory can be considered as two-part objects. The operations used on each part are different:

- To find the size of the pointer part, use `sizeof`; for the array part use `strlen()`.
- Remember to use the `==` operator when comparing the pointer parts, but use `strcmp()` to compare the array parts. When using `strcmp()` remember to compare the result to 0 when testing for equality.
- To copy the pointer part of a string, use `=`; to copy the array part, use `strcpy()` or `strncpy()`. Make sure the destination array is long enough to hold the result.
- The following table summarizes the syntax for initializing, assigning, or copying a string:

Operation	With a <code>char</code> Array	With a <code>char*</code>
Initialization	<code>char word[10] = "Hello";</code>	<code>char* message = word;</code>
Assignment	Does not copy the letters <code>Cannot do word = "circle"</code>	<code>char* message = "square";</code> <code>message = "circle";</code>
String copy	<code>strcpy( word, "Bye" );</code>	Error: <code>strcpy( message, "Thanks" );</code> Cannot change a string literal.
String comparison	<code>strcmp( word, "Bye" )</code>	<code>strcmp( message, "Bye" )</code>

**Quoting.** A string containing one character is not the same as a single character. The string also contains a null terminator. Be sure to write single quotes when you want a character and double quotes when you want a string. The double quotes tell C to append a null terminator. You can embed quotation marks within a string, but you must use an escape character. Also, be careful to end all strings properly, or you may find an unwanted comment in the output.

#### 12.6.4 New and Revisited Vocabulary

These are the most important terms and concepts presented or discussed in this chapter:

<code>string</code>	NULL pointer	parallel arrays
<code>string literal</code>	conversion specifier	C <code>string</code> library
<code>string merging</code>	right and left justification	array of strings
<code>two-part object</code>	string variable	ragged array
<code>null terminator</code>	string assignment	menu functions
<code>null character</code>	string comparison	buffer overflow
<code>null string</code>	string concatenation	menu selection validation

The following keywords and C library functions are discussed in this chapter.

<code>\\" and \0</code>	<code>gets()</code>	<code>strncat()</code>
<code>char *</code>	<code>puts()</code>	<code>strchr()</code>
<code>typedef</code>	<code>strlen()</code>	<code>strrchr()</code>
<code>char []</code>	<code>strcmp()</code>	<code>strstr()</code>
<code>printf() %s conversion</code>	<code>strncmp()</code>	<code>sizeof</code> a string
<code>scanf() %ns conversion</code>	<code>strcpy()</code>	<code>==</code> (on strings)
<code>scanf %n[^?]</code> conversion	<code>strncpy()</code>	<code>=</code> (on strings)
	<code>strcat()</code>	

The following keywords and C++ library functions are discussed in this chapter.

<code>string</code>	<code>cin &gt;&gt;</code>	<code>replace()</code>
<code>==</code> (on strings)	<code>cout &lt;&lt;</code>	<code>append()</code>
<code>=</code> (on strings)	<code>getline()</code>	<code>find_first_of()</code>
<code>nullptr</code>	<code>get()</code>	<code>find_last_of()</code>
<code>length()</code>		<code>find()</code>

### 12.6.5 Where to Find More Information

- Dynamically allocated memory can be used to initialize a string variable. This is explored in Chapter 16.
- Pointers are introduced in Chapter 11 and the use of pointers to process arrays is explained in Chapter 16.
- A variety of effects can be implemented using the `scanf()` brackets conversion. The interested programmer should consult a standard reference manual for a complete description.
- Chapter 8 discusses whitespace in the input stream, the problems it can cause, and how to use `scanf()` with `%c` to handle these problems.



## Chapter 13

# Enumerated and Structured Types

We have discussed two kinds of aggregate data objects so far: arrays and strings. This chapter introduces a third important **aggregate type**: structures<sup>1</sup>.

Structures provide a coherent way to represent the different properties of a single object. A **struct** variable is composed of a series of slots, called **members**, each representing one property of the object. Unlike an array, in which all parts have the same type, the components of a **structure** generally are of mixed types. Also, while the elements of an array are numbered and referred to by a subscript, the parts of a structure are given individual names. We examine how structures are represented, which operations apply to them, and how to combine arrays with structures in a compound object.

Enumerations are used to create symbolic codes for collections of conditions or objects. They can be useful in a variety of situations, such as handling error conditions, where they list and name the possible types of outcomes.

### 13.1 Enumerated Types

The use of **#define** to give a symbolic name to a constant is familiar by now. It is a simple way to define a symbolic name for an isolated constant such as  $\pi$  or the force of gravity. Sometimes, though, we need to define a set of related symbols, such as codes for the various kinds of errors that a program can encounter. We wish to give the codes individual names and values and declare that they form a set of related items. The **#define** command lets us name values individually but gives us no way to associate them into a group of codes whose meaning is distinct from all other defined symbols. For example, suppose we wanted to have symbolic names for the possible ways that a data item could be illegal. We could define the following four symbolic constants:

```
#define DATA_OK 0
#define TOO_SMALL 1
#define TOO_BIG 2
#define NO_INPUT -1
```

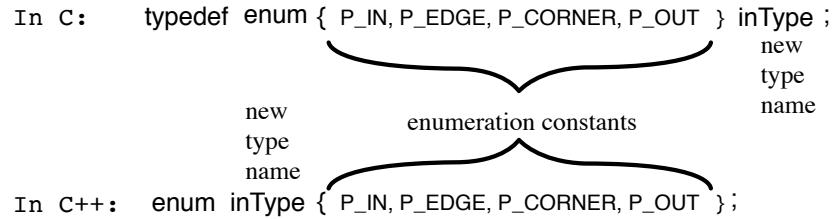
Enumerated types were invented to address this shortcoming. They allow us to define a type name that groups together a set of related symbolic codes.

#### 13.1.1 Enumerations

The character data type really is an enumeration. The ASCII code maps the literal forms of the characters to corresponding integer values. This particular enumeration is built into the C language, so you do not see its construction. But it is possible to create new types using the **enumeration specification**, the keyword **enum** followed by a bracketed list of symbols called **enumeration constants**. Enumerations normally are defined within a **typedef** declaration, as shown in Figure 13.1.

---

<sup>1</sup>Union data types are similar to structures with more than one possible configuration of members. They are not covered in this text because their applications are better covered by polymorphic types in modern languages. Applications that require such types are better programmed in C++ or Java.



The underlying values of `P_IN`...`P_OUT` are 0, 1, 2, 3 respectively.

**Figure 13.1. The form of an `enum` declaration in C and C++.**

Although the `enum` and `#define` mechanisms produce similar results, using an `enum` declaration is superior to using a set of `#define` commands because we can declare that the enumeration codes listed form the complete set of relevant codes for a given context. An enumerated type can, therefore, define the set of valid values a variable may have or a function may return. Using `enum` rather than `#define` improves readability with no cost in efficiency.

The enumeration symbols will become names for constants, much as the symbol in a `#define` becomes a name for a constant. By default, the first symbol in the enumeration is given the underlying value 0; succeeding symbols are given integer values in ascending numeric order, unless the sequence is reset by an explicit assignment. Such an assignment is illustrated by the third declaration in Figure 13.2. An explicit initializer resets the sequence, which continues from the new number, so it is possible to leave gaps in the numbering or have two constants represented by the same code, which normally is undesirable. Therefore, the programmer must be careful when assigning specific representations to enumeration constants.

Inside a C program, enumeration values are treated as integers in all ways, as is true for characters. The compiler makes no distinction between a value of an enumeration type and a value of type `int`. Since enumeration constants are translated into integers, all the integer operators can be used with enumeration values. Although most integer operators make no sense with enumeration constants, increment and decrement operators sometimes are used, as they are with type `char`, to compute the next value in the code-series. However, unlike `char`, C provides no way to automatically read or print enumeration values; they must be input and output as integers. This complication leads to extra work to devise special input and output routines for values of an enumeration type. The extra hassle sometimes discourages the programmer from using an `enum` type. However, the added clarity in a program normally is worth the effort, and in truth, the values of many such types never actually are input or output, but merely used within the program to send information back and forth between functions.

In a C++, program, enumerated types are distinct from integers, but can be converted to and from integers by casting.

**Why use enumerations?** We use enumerations (rather than integer constants) to distinguish each kind of code from ordinary integers and other codes. This enables us to write programs that are less cryptic. Someone reading the program sees a symbol that conveys its meaning better than an integer that encodes the meaning. The reader understands the program more easily because the use of `enum` clarifies which symbols are related to each other. By using an `enum` constant rather than its associated integer, we make it evident that the object is

---

```
typedef enum { P_IN, P_SIDE, P_CORNER, P_OUT } inType;
typedef enum { NO_ROOT, ROOT_OK } statusT;
typedef enum { DATA_OK, TOO_SMALL, TOO_BIG, NO_INPUT=-1 } errorT;

enum inType { P_IN, P_SIDE, P_CORNER, P_OUT };
enum statusT { NO_ROOT, ROOT_OK };
enum errorT { DATA_OK, TOO_SMALL, TOO_BIG, NO_INPUT=-1 };
```

---

**Figure 13.2. Four enumerations in C and again in C++.**

---

We declare an enumeration variable and use it to store an error code after testing an input value.

```
double x;
errorT status;

scanf( "%lg", &x );
if (x < 0) status = TOO_SMALL;
else status = DATA_OK;
```

---

**Figure 13.3. Using an enumeration to name errors.**

a code, not a number.

**Notes on Figure 13.2. Four enumerations.** This figure declares three enumerated types that give names to conditions that are somewhat complex to characterize but easy to understand once they are named. These types will be used in various programs in the remainder of the text.

**Type in\_type: answer codes.** The first `typedef` declares a set of codes that will be used in an application in this Chapter. This program processes a point and a rectangle in the  $xy$  plane, determining where the point is located with respect to the rectangle. One function does the analysis and returns an answer of type `in_type` to the main program, which then interacts with the user. The code `P_IN` is used if the point is inside the rectangle; the codes `P_SIDE`, `P_CORNER`, and `P_OUT`, respectively, mean that the point is on a side, on a corner, or outside of the rectangle.

**Type status\_t: status codes.** The second declaration in Figure 13.2 is an enumeration of codes used in a program that finds the roots of an equation (Figure 13.7). In this program, the main program prompts the user for equation parameters, then calls a function to do the actual work. The function returns the root (if found) and a success-or-failure code, `ROOT_OK` or `NO_ROOT`. The descriptive nature of these symbols makes the program logic clearer.

**Type error\_t: error codes.** The third declaration in Figure 13.2 is an enumeration of **error codes**, which can be used to simplify the management of input errors. The symbols `DATA_OK`, `TOO_SMALL`, and `TOO_BIG` carry the underlying integer values 0, 1, and 2, respectively. The last symbol, `NO_INPUT`, is followed by an assignment, so the symbol `NO_INPUT` will be represented by the code `-1` rather than 3. Figure 13.3 illustrates a code fragment that applies this enumerated type. These statements test for an input error and set the value of a variable to indicate whether it was detected. Further input error processing is described in Chapter 14.

### 13.1.2 Printing Enumeration Codes

Since enumeration constants are represented by integers in C, we could print them as integers. However, the purpose of using an enumeration is to give meaningful symbolic names to status codes, so we certainly do not want to see a cryptic integer code in the output. Rather, we want to see an English word.

Symbolic output is achieved easily using a ragged array. An array of strings is declared that is parallel to the enumeration. Each word in the array is the English version of the corresponding symbol in the enumeration. We use the symbol (an integer) to subscript the ragged array and select the corresponding string, which then is used in the output. The array of strings is defined in the same program scope as the enumeration, so that any function that uses the enumeration has access to the output strings. Since enumerations usually are defined globally, the string array also is global. An important precaution with any global array is to begin the declaration with the keyword `const` so that no part of the program code can change it accidentally. As an example, consider an enumeration defined in Chapter 8:

```
typedef enum { P_IN, P_SIDE, P_CORNER, P_OUT } in_type;
```

We can define output strings thus:

```
const char* in_labels[] = { "inside", "on a side of", "on a corner of", "outside" };
```

To show how to use this array of strings, assume we have a variable, `position`, that contains a value of type `in_type`. We write the following lines to insert the English equivalent of the value of `position` into the middle of a sentence and display the result on the screen:

We diagram the storage allocated for the quadratic root program in Figures 13.5 and 13.6. The diagram on the left shows the contents of memory just after calling the `solve()` function and before its code is executed. The diagram on the right shows the situation during the function return process, just before the function's storage disappears.

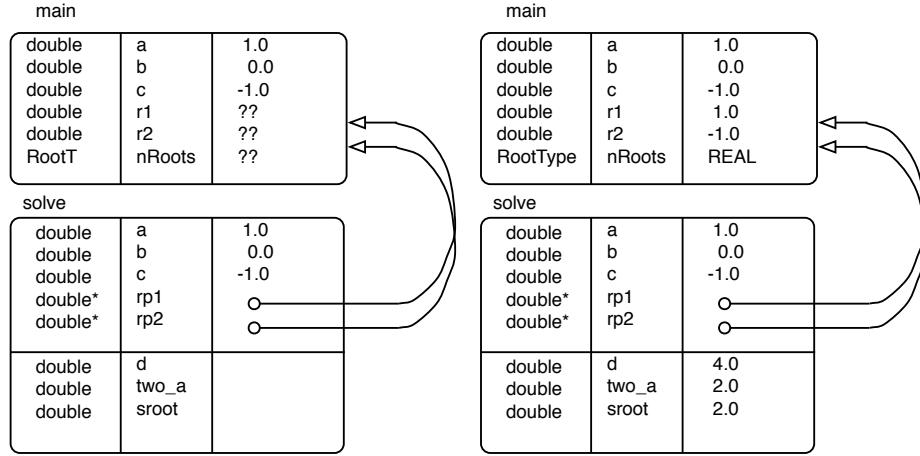


Figure 13.4. Memory for the quadratic root program.

```
in_type position;
...
printf( " The point is %s the square.\n", in_labels[position] );
```

The output might look like this

```
The point is on a corner of the square.
```

### 13.1.3 Returning Multiple Function Results

The program in Figures 13.5 through 13.7 illustrate a typical use of an enumerated type and the use of call by address to return multiple results from a function. The enumerated type helps to make this code very straightforward and readable.

The type definition and main program (in C, Figure 13.5 then in C++, Figure 13.6, does only user interaction, input and output. The equation is solved by the function in Figure 13.7 which must then return one, two, or three results, depending on the coefficients of the equation. This code is the same for both C and C++.

---

```

#include <stdio.h>
#include <math.h>
#include <stdbool.h>

typedef enum ZERO, NONE, LINEAR, SINGLE, REAL, COMPLEX RootT;

RootT solve( double a, double b, double c, double* rp1, double* rp2 );
// -----
int main( void )
{
    double a, b, c;                      // Coefficients of the equation.
    double r1, r2;                        // Roots of the equation.
    RootT nRoots;                         // How many roots the equation has.

    puts( "\n Find the roots of a*x^2 + b*x + c = 0" );
    printf( " Enter the values of a, b, and c: " );
    scanf( "%lg%lg%lg", &a, &b, &c );
    printf( "\n The equation is %.3g *x^2 + %.3g *x + %.3g = 0\n", a, b, c );

    nRoots = solve( a, b, c, &r1, &r2 );
}

switch (nRoots) {
    case ZERO:
        printf( "\n Degenerate equation -- a = b = c = 0.\n\n" );
        break;
    case NONE:
        printf( "\n Degenerate equation -- no roots\n\n" );
        break;
    case LINEAR:
        printf( "\n Linear equation -- root at -c/b = %.3g\n\n", r1 );
        break;
    case SINGLE:
        printf( "\n Single real root at x = %.3g \n\n", r1 );
        break;
    case REAL:
        printf( "\n The real roots are x = %.3g and %.3g \n\n", r1, r2 );
        break;
    case COMPLEX:
        printf( "\n The complex roots are x = %.3g + %.3g i and "
               "x = %.3g - %.3g i \n\n", r1, r2, r1, r2 );
}
}

return 0;
}

```

---

Figure 13.5. Solving a quadratic equation in C.

**Figure 13.6.** Solving a quadratic equation in C++.**Notes on Figures 13.5 and 13.6: Using the quadratic formula to solve an equation.**

**First box, Figure 13.5 and lines 6 of Figure 13.6:** status code enumeration of cases. We define an enumerated type to represent the five possible ways in which a quadratic equation can have or not have roots.

1. **NONE** means the equation is degenerate because the coefficients are all zero.
2. **NONE** means the equation is inconsistent; it has a nonzero constant term but no variable terms.
3. **LINEAR** means the equation is linear because the coefficient of  $x^2$  is 0.
4. **SINGLE** means the two real roots of this equation are the same; if graphed, the function would be tangent to the  $x$  axis at the root value.
5. **REAL** means there are two real roots; if graphed, the function would cross the  $x$  axis twice.
6. **COMPLEX** means there are two complex roots; if graphed, the function would not touch or cross the  $x$  axis.

**Second box, Figure 13.5 and lines 7 and 22 of Figure 13.6:** calling the `solve()` function. This function call takes three arguments into the function (the three coefficients of the equation), passing them by value. It also supplies two addresses in which to store the roots of the equation, should they exist. The value returned directly by the function will be one of the status codes from the enumeration in the first box. In Figure 13.4, note that the values of the first three arguments have been copied into the memory area of the function, but addresses (pointers) are stored in the last two parameters.

**Third box, Figure 13.5 and lines 23–42 of Figure 13.6:** interpreting the results. The status code returned by the function is stored in `nRoots`. We use a `switch` statement to test the code and select the appropriate format for displaying the answers. It is quite common to use a `switch` statement to process an enumerated-type result because it is a simple and straightforward way to handle a complex set of possibilities. It is not necessary to use a default case, since all of the enumerated type's constants have been covered. This programming technique also has the advantage that it separates the user interaction from the algorithm that computes the answers, making both easier to read and write.

**Output.**

```
Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 1 0 -1
The equation is 1 *x^x + 0 *x + -1 = 0

The real roots are x = 1 and -1
```

**Notes on Figures 13.7 and 13.4. The quadratic root function.** This computational function is the same for C and C++.

**First box: A zero test.**

- We use `double` values for the coefficients of the equation. To make the program more robust and useful in various circumstances, we use an epsilon test for all comparisons. We arbitrarily choose an epsilon value of  $10^{-100}$ , which is small enough to be virtually zero but large enough to ensure that using it as a divisor will not cause floating point overflow.
- The `iszero()` function uses the `EPS` value to compare its argument to 0. We write the code as a separate function to remove repetitive clutter from the `solve()` function, making it shorter and clearer.
- This function returns a `bool` result, which can be used directly in an `if` statement or a logical expression.

***Second box: degenerate cases.***

- We test for coefficients that do not represent quadratic equations and return the appropriate code from the enumerated type. Some sort of test for zero is necessary to avoid the possibility of division by zero or near zero in the third and fourth boxes. We use a tolerance of  $10^{-100}$ .
- By using the `if...return` control structure, we eliminate the need for `else` clauses and nested conditionals. The added logic needed to support a single return statement would be a considerable distraction from the simplicity of this solution.
- Error output:

```
Find the roots of a*x^2 + b*x + c = 0
```

This function is called from the quadratic root program in Figures 13.5 and 13.6. One result is returned by a `return` statement; the other two (if they exist) are returned through pointer parameters.

```
#define EPS 1e-100
bool iszero( double x ) { return fabs( x ) < EPS; }

RootT      // Return value is the enum constant for number and type of roots.
solve( double a, double b, double c, double * rp1, double * rp2 )
{
    double d, two_a, sroot;           // Working storage.

    if (iszero( a ) && iszero( b )) { // Degenerate cases.
        if (iszero( c )) return ZERO;
        else return NONE;
    } if (iszero( a )) {
        *rp1 = -c / b;
        return LINEAR;
    }

    two_a = 2 * a;
    d = b * b - 4 * a * c;          // discriminant of equation.
    if (iszero( d )) {              // There is only one root.
        *rp1 = -b / two_a;
        return SINGLE;
    }

    sroot = sqrt( fabs( d ) );       // fabs is floating point absolute value.
    if (d > EPS) {                 // There are 2 real roots.
        *rp1 = -b / two_a + sroot / two_a;
        *rp2 = -b / two_a - sroot / two_a;
        return REAL;
    }
    else {                         // There are 2 complex roots.
        *rp1 = -b / two_a;
        *rp2 = sroot / two_a;
        return COMPLEX;
    }
}
```

Figure 13.7. The quadratic root function for both C and C++.

```
Enter the values of a, b, and c: 0 0 0
The equation is 0 *x^x + 0 *x + 0 = 0
```

```
Degenerate equation -- a = b = c = 0.
```

---

```
Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 0 0 1
The equation is 0 *x^x + 0 *x + 1 = 0
```

```
Degenerate equation -- no roots
```

---

```
Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 0 3 1
The equation is 0 *x^x + 3 *x + 1 = 0
```

```
Linear equation -- root at -c/b = -0.333
```

---

*Third box: A single root.*

- We compute and store the values of `two_a` and `d` because these subexpressions are used several times in this box and the next. Factoring out these computations increases run-time efficiency and shortens the code.
- If the equation is quadratic, it can have two separate roots, real or complex, or one repeated real root. We look at the discriminant of the equation, `d`, to identify how many roots are present. If `d` equals 0, the equation has only one distinct root, which we then compute and return through the first pointer parameter.
- The function return value tells the main program that only one of the pointer parameters has been given a value (the other is unused).
- Single root output:

```
Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 2 4 2
The equation is 2 *x^x + 4 *x + 2 = 0
```

```
Single real root at x = -1
```

*Fourth box: Two different roots.*

- We compute `sroot` once and use the stored value to compute the roots in each of the two remaining cases.
- In both cases, two `double` values are returned to `main()` by storing them indirectly through the pointer parameters. The function return value tells `main()` how to interpret these two numbers.
- The `REAL` case is illustrated in Figure 13.4 and the corresponding output is shown in the notes for the main program. Note that the values of `r1` and `r2` in `main()` have been changed by storing numbers indirectly through the two pointer parameters.
- Output with two roots, real or complex:

```
Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 1 0 -1
The equation is 1 *x^x + 0 *x + -1 = 0
```

```
The real roots are x = 1 and -1
```

---

```
Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 1 0 1
The equation is 1 *x^x + 0 *x + 1 = 0
```

```
The complex roots are x = -0 + 1 i and x = -0 - 1 i
```

---

```

tag name
typedef struct BOX { int length, width, height;
                      float weight;
                      ...
                      char contents[32];
} BoxT;
typedef name

```

member fields

---

Figure 13.8. Modern syntax for a C struct declaration.

## 13.2 Structures in C

This section presents structures in C and explains the syntax used to do operations on structures. The following section builds on these explanations and shows how to use structures and classes in C++.

A structure type specification starts with the keyword **struct**, followed by an optional identifier called the **tag name**, followed by the member declarations enclosed in curly brackets. (Members also may be called *fields* or *components*.) All this normally is written as part of a **typedef** declaration, which gives another name, the **typedef name**, to the type. The members can be almost any combination of data types, as in Figure 13.8. Members can be arrays or other types of structures. However, a structure cannot contain a member of the type it is defining (the type of a part cannot be the same as the type of the whole).

**New types.** A new type name can be defined using **typedef** and **struct**, (See Figure 13.8.) and may be used like a built-in type name anywhere a built-in type name may be used. Important applications are to

1. Declare a structured variable.
2. Create an array of structures.
3. Declare the type of a function parameter.
4. Declare the type of a function result.

Before the **typedef** was introduced into C, the tag name was the only way to refer to a structured type. Using tag names is syntactically awkward because the keyword **struct** is part of the type name and must be used every time the tag name is used. For example, a type definition and variable declaration might look like this:

```
struct NAME { char first[16]; char last[16]; } boy;
```

or they could be given separately, like this:

```
struct NAME { char first[16]; char last[16]; };
struct NAME boy;
```

Initially, this was addressed in C by introducing the **typedef** declaration. In C++, however, the problem was solved properly. Now, with either a **struct** or a **class**, the name following the keyword is the name of the type.

Today, tag names are not used as much in C because **typedef** is less error prone and serves most of the purposes of tag names. A tag name is optional when a **typedef** declaration is given and the **typedef** name is used. Tag names have two remaining important uses. If one is given, some on-line debuggers provide more informative feedback. They also are useful for defining recursive types such as trees and linked lists.<sup>2</sup> When present, the tag name should be related to the **typedef** name. Naming style has varied; the most recent seems to be to write tag names in upper case and **typedef** names in lower case. We supply tag names with no further comment for the types defined in this chapter.

The result of a **typedef struct** declaration is a “pattern” for creating future variables; it is not an object and does not contain any data. Therefore, you do not include any initializers in the member declarations.

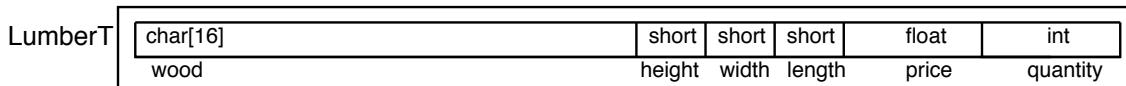
---

<sup>2</sup>In such types, one member is a pointer to an object of the **struct** type currently being defined. These topics are beyond the scope of this text.

A **struct** type specification creates a type that can be used to declare variables. The **typedef** declaration names this new type.

```
typedef struct LUMBER {           // Type for a piece of lumber.
    char wood[16];                // Type of wood.
    short int height;              // In inches.
    short int width;               // In inches.
    short int length;              // In feet.
    float price;                  // Per board, in dollars.
    int quantity;                 // Number of items in stock.
} LumberT;
```

The members of **LumberT** will be laid out in memory in this order. Depending on your machine architecture, padding bytes might be inserted between member fields.



**Figure 13.9. A struct type declaration in C.**

**Structure members.** Member declarations are like variable declarations: Each consists of a type, followed by a list of member names ending with a semicolon. Any previously defined type can be used. For example, a structure listing the properties of a piece of lumber is defined in Figure 13.9. We represent the dimensions of the board as small integers (a  $2 \times 4 \times 8$  board is cut at the sawmill with a cross section of approximately  $2 \times 4$  inches and a length of 8 feet). We use the types **float** to represent the price of the lumber and **int** to represent the quantity of this lumber in stock. Figure 13.9 also shows a memory diagram of the structure created by this declaration.

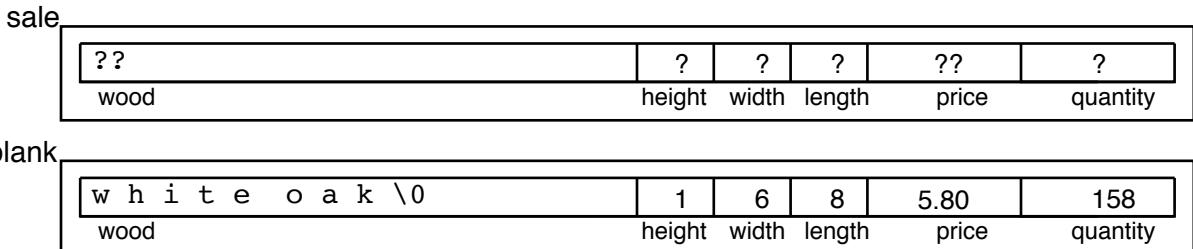
The **member names** permit us to refer to individual parts of the structure. Each name should be unique and convey the purpose of its member but do so without being overly wordy. (This is good advice for any variable name.) A member name is used as one part of a longer compound name; it needs to make sense in context but need not be complete enough to make sense if it stood alone.

A component **selection expression** consists of an object name followed by one or more member names, all separated by the dot operator. Because such expressions can get lengthy, it is a good idea to give each member a brief name.

---

We use the type declared in Figure 13.9.

```
LumberT sale;
LumberT plank = { "white oak", 1, 6, 8, 5.80, 158 };
int bytes = sizeof (LumberT);
```




---

**Figure 13.10. Declaring and initializing a structure.**

**Declaring and initializing a C structure.** We can declare a structured variable and, optionally, initialize it in the declaration. The two declarations in Figure 13.10 use the structured type declared in Figure 13.9 to declare variables named `sale` and `plank`. The variable `sale` is uninitialized, so the fields will contain unknown data. The variable `plank` is initialized. The **structured variable initializer** is much like that for an array. It consists of curly brackets enclosing one data value, of the appropriate type, for each member of the structure. These are separated by commas and must be placed in the same order as the member declarations. The arrangement of the values in `plank` is diagrammed in the figure. Like an array, if there are too few initializers, any remaining member will be filled with 0 bytes.

## 13.3 Operations on Structures

ISO C supports a variety of **operations on structures**:<sup>3</sup> Standard ISO C compilers permit the following operations:

1. Find the size of the structure.
2. Set a pointer's value to the address of a structure.
3. Access one member of a structure.
4. Use assignment to copy the contents of a structure.
5. Return a structure as the result of a function.
6. Pass a structure as an argument to a function, by value or address.
7. Include structures in larger compound objects, such as an array of structures.
8. Access one element in an array of structures.
9. Access one member of one structure in an array of structures.

One important and basic operation is not supported in C: comparison of two structures. When comparison is necessary, it must be implemented as a programmer-defined function. The remainder of this section isolates and explains each of these operations. Figures 13.14, 13.15, 13.16, and 13.18 contain function definitions that are part of the complete program in Figures 13.20 and 13.21.

### 13.3.1 Structure Operations

**The size of a structure.** C was designed to be used on many kinds of computers, with various hardware characteristics. Since the language was introduced in 1971, hardware has changed drastically and is still changing. The language standard specifies how a program should behave, but not how that program should be represented in machine language. The C standard states that the members of a structure must be stored in order, but it permits extra bytes, or *padding* to be inserted between members. Thus, some structures occupy more total bytes than the sum of the sizes of their members. When padding is used, the extra bytes are inserted by the compiler into the structure to force each structure member to start on a memory address that is a multiple of 2 (for small computers) or 4 (most modern computers). This process is called *alignment* and it is done to match hardware requirements or to improve efficiency.

Padding is only used after small or irregular-sized members. It is not an issue with types `double`, `float`, `long`, or `int`, because these types already meet or exceed all hardware alignment requirements. Single characters and odd-sized strings, however, will likely be followed by padding. In the `LumberT` structure, the first member is a `char` array of length 11. It will be padded by 1 byte in any modern compiler. The next three members are type `short` integers. Some compilers might pad each one to four bytes, and some compilers will not add padding at all. Most current compilers will insert two bytes of padding after the third `short int`, because the `float` that follows it must be aligned to an address that is a multiple of 4. Thus, the size of the structure pictured in Figure 13.10 could be as little as 24 bytes (if `sizeof(int) = 2`) or as great as 32 bytes, (if `sizeof(int) = 4` and all possible padding was added). In the `gcc` compiler, it is 28 bytes.

---

<sup>3</sup>Older, pre-ANSI C compilers may not support some operations, particularly assignment. Therefore, what you are permitted to do with a structure depends on the age of your compiler.

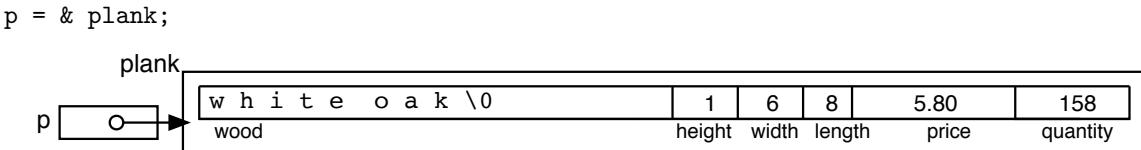


Figure 13.11. Set and use a pointer to a structure.

In most situations, the programmer does not need to know whether padding is used by the compiler. Sometimes, however, a program must manage its own memory, and needs to know how large a structure is in order to allocate memory space for it<sup>4</sup>. `sizeof` operator is used for this purpose, as shown in the first line in Figure 13.10.

**Set a pointer to a structure.** We can declare a pointer and set its value to the address of a `struct`. In Figure 13.11, the address of the structure named `plank`, defined in Figure 13.10, is stored in the pointer variable `p`. The address used is the address of the first member of the structure, but the pointer gives access to the entire structure, as illustrated.

**Access one member of a structure.** The members of a structure may be accessed either directly or through a pointer. The **dot** (period) **operator** is used for direct access. We write the variable name followed by a dot, followed by the name of the desired member. For example, to access the `length` member of the variable `plank`, we write `plank.length`. Once referenced, the member may be used anywhere a simple variable of that type is allowed.

The **arrow operator** is used to access the members of a structure through a pointer. We write the pointer name followed by `->` (arrow) and a member name. For example, since the pointer `p` is pointing at `plank`, we can use `p` to access one member of `plank` by writing `p->price`. This is illustrated by the last line in Figure 13.11. We use both access operators in Figure 13.12 to print two members of `plank`. The output from these two print statements is

```
Plank price is $5.80
Planks in stock: 158
```

**Structure assignment.** ISO compilers let us copy the entire contents of a structure in one assignment statement. A structure assignment copies all of the values of the members of the structure on the right into the corresponding members of the structure variable on the left (which must be the same type).<sup>5</sup> Figure 13.13 gives an example.

### 13.3.2 Using Structures with Functions

Three functions are given in this section to illustrate how structures can be passed into and out of functions. Calls on these functions are incorporated into the program in Figures 13.20 and 13.21.

**Returning a structure from a function.** The same mechanism that allows us to do structure assignment lets us return a structure as the result of a function. For example, in Figure 13.14, we declare a function, `read_lumber()`, that reads the data for a piece of lumber into the members of a local structured variable. In

<sup>4</sup>This is explained in Chapter 16.

<sup>5</sup>This actually is a “shallow” copy. If one of the members is a pointer to an object, assignment makes a copy of the address but not the object it references. Then two objects refer to a common part and any change in that part is “seen” in both copies! Deep copies, which duplicate both pointer and referent, are beyond the scope of this text.

---

```
printf( "%s price is $%.2f\n", plank.wood, plank.price ); // direct access
printf( "%s in stock: %i\n", p->wood, p->quantity ); // indirect access
```

---

Figure 13.12. Access one member of a structure.

---

sale (before assignment)

??	?	?	?	??	?
wood	height	width	length	price	quantity

sale (after assignment)

w h i t e o a k \0	1	6	8	5.80	158
wood	height	width	length	price	quantity

---

**Figure 13.13.** Structure assignment.

these `scanf()` statements, we do not need parentheses around the entire member name. The dot operator has higher precedence than the address operator, so we get the address of the desired member. When all input has been read, we return a copy of the contents of the local structure as the function's value<sup>6</sup>.

Guidance: This is yet another way to return multiple values from a function, but should be avoided if the structure has more than a few members because of the large amount of copying involved. For large structures, call by address should be used instead.

**Call by value with a structure.** We can pass a structure as an argument to a function. When we use call by value to do this, all the members of the caller's argument are copied into the members of the function's parameter variable, just as if each were a separate variable. Thus, the technique is used primarily for small structures and call by const address is preferable for structures with more than a few members.

In Figure 13.15, we declare a function, `print_lumber()` with a structure parameter. It formats and prints

<sup>6</sup>Note that we are returning a copy of the local variable, not a pointer to it.

---

```
LumberT read_lumber( void )
{
    LumberT board;
    printf( " Reading a new stock item. Enter the three dimensions of a board: " );
    scanf( "%hi%hi%hi", &board.height, &board.width, &board.length );
    printf( " Enter the kind of wood: " );
    scanf( "%15[^\\n]", board.wood );
    printf( " Enter the price: $" );
    scanf( "%g", &board.price );
    printf( " Enter the quantity in stock: " );
    scanf( "%i", &board.quantity );
    return board;
}
```

---

**Figure 13.14.** Returning a structure from a function.

---

```
void print_lumber( LumberT b )
{
    printf( " %s %hi\" x %hi\" x %hi feet long -> %i in stock at $%.2f\\n",
            b.wood, b.height, b.width, b.length, b.quantity, b.price );
}
```

---

**Figure 13.15.** Call by value with a structure.

---

```

void sell_lumber( int sold, LumberT* board )
{
    if (board->quantity >= sold) {
        board->quantity -= sold;
        printf( " OK, sold %i\n", sold %i %s\n", sold, board->wood );
        print_lumber( *board );
    }
    else printf( " Error: cannot sell %i %s boards; have only %i.\n",
                  sold, board->wood, board->quantity );
}

```

---

**Figure 13.16.** Call by address with a structure.

the data in the structure. Because the parameter is a structure (not a pointer), we use the dot operator to access the members. Calls on `read_lumber()` and `print_lumber()` might look like this.

```

sale = read_lumber();
print_lumber( sale );

```

The input would be read into a local variable in the `read_lumber()` function, then copied into `sale` when the function returns. It would be copied a second time into parameter `b` in `print_lumber()`. Calls on `read_lumber()` and `print_lumber()` using pointers might look like this:

```

*p = read_lumber(); /*// Store a structured return value.
print_lumber( *p ); /* // Echo-print stock[3].

```

The output produced by this interaction might be

```

Reading a new stock item.
Enter the three dimensions of a board: 2 3 6
Enter the type of wood: fir
Enter the price: $4.23
Enter the quantity in stock: 16
fir      2" x 3" x 6 feet $4.23 stock: 16

```

**Call by address with a structure.** We can pass a pointer to a structure, or the address of a structure, as an argument to a function. This, **call by address**, gives us more functionality than call by value. In Figure 13.16, we declare a function named `sell_lumber()` that takes an integer (the item number) and a structure pointer as its parameters and checks the inventory of that item. If the supply is adequate, the quantity on hand is reduced by the number sold; otherwise an error comment is printed. The following sample calls illustrate two ways to call this function:

```

sell_lumber( p, 200);           // A pointer argument.
sell_lumber( &stock[1], 2 );

```

The output would look like this:

```

Error: cannot sell 200 boards (only have 158).
OK, sold 2
fir      2" x 3" x 6 feet $4.23 in stock: 14

```

On the first line of `sell_lumber()`, we use the parameter (a pointer) with the arrow operator to access a member of the structure. Within the code is a call on `print_lumber()` to show the reduced quantity on hand after a sale. Note that the pointer parameter of `sell_lumber()` is the wrong type for `print_lumber()`, because the argument for `print_lumber()` must be an item, not a pointer. We overcome this incompatibility by using a dereference operator in the call to `print_lumber()`. The result of the dereference is a structured value.

**Value vs. address parameters.** The three functions just discussed, as well as the `boards_equal()` function in Figure 13.18 have been implemented using call by value wherever possible; it was necessary to use a pointer parameter (call by address) for `sell_lumber()`, because the function updates its parameter to return information to the caller. The prototypes are

```
LumberT read_lumber( void );
void print_lumber( LumberT );
void sell_lumber( int, LumberT* );
bool boards_equal( LumberT, LumberT );
```

Pointer parameters also could have been used in `print_lumber()`, `read_lumber()`, and `boards_equal()`. This would have the advantage of reducing the amount of copying that must happen at run time because only a pointer, not an entire structure, must be copied into the function's parameter. The time and space needed for copying can be significant for large structures. The same is true of the time consumed by returning a structured result from a function like `read_lumber()`. If the structure has many members or contains an array of substantial size as one member, it usually is better to use call by address to send the information to a function and to return information.

In any case, call-by-address requires extra care because it does not automatically isolate the caller from the subprogram. Thus, the subprogram can change the value stored in the caller's variable. In functions like `print_lumber()` and `boards_equal()` that use, but do not modify, their parameters, a `const` should be added to the parameter type to prevent accidental mistakes that are hard to track and correct. The revised prototypes for a large structure in this program would be

```
void read_lumber( LumberT* );
void print_lumber( const LumberT* );
void sell_lumber( int, LumberT* );
bool boards_equal( const LumberT*, const LumberT* );
```

One disadvantage of using a pointer parameter is that every reference to every member of the structure is *indirect*; first the reference address must be fetched from the parameter, then it must be used to access the memory location of the needed member. If members of the structure are used many times in the function, these extra references can substantially slow down execution, and it is better to use call by value.

Finally, when call by value is used, a copy is made. Sometimes the copy plays an important part in the program. For example, when modifying the data in one record of a database, it is good engineering practice to permit the process to be aborted midway. To do this, all modifications are made to a local *copy* of the data, and the original remains unchanged until the modification process is finished and the user confirms that the result is acceptable. At this time, the changes are *committed*, and the revised data record is returned to the caller and copied into the location of the original. In actual practice, this is a very important technique for maintaining a database of any sort, with or without an underlying database engine.

### 13.3.3 Arrays of Structures

It is possible to include one aggregate type within another. In the definition of the `struct` for `LumberT`, we include a character array. A `struct` type also may be the base type of an array. Thus, we can declare and, optionally, initialize an **array of structures**. An initializer for an array of structures is enclosed in curly brackets and contains one bracketed set of values for each item in the array, separated by commas.<sup>7</sup> An example and its diagram are shown in Figure 13.17, where we declare and initialize an array of `LumberT` structures.

**Accessing one structure in an array of structures.** The name of the array and a subscript are needed to access one element in an array of structures, shown in the following code. The first line assigns a structured value to the last item of an array, the second calls `print_lumber()` to display the result. To send one structure from the array, by address, to a function that has a structure pointer parameter, the argument needs both a subscript and an ampersand, as shown by the call on `sell_lumber()` on the third line.

```
stock[3] = read_lumber();      // Read into one element.
print_lumber( stock[3] );    // Print one element.
sell_lumber( &stock[3], 2 );  // Call by address.
```

<sup>7</sup>In addition to the commas between members, it is legal to have a comma after the last one. This sometimes makes it easier to create large data tables with a word processor, without having to remember to remove the comma from the last one.

```
LumberT stock[5] = { { "spruce", 2, 4, 12, 8.20, 27 },
                     { "pine", 2, 3, 10, 5.45, 11 },
                     { "pine", 2, 3, 8, 4.20, 35 },
                     { "" // Remaining members will be set to 0 }
};

stock
```

[0]	s p r u c e \0 wood	2	4	12	8.20	27
[1]	p i n e \0 wood	2	3	10	5.45	11
[2]	p i n e \0 wood	2	3	8	4.20	35
[3]	\0 wood	0	0	0	0.00	0
[4]	\0 wood	0	0	0	0.00	0

Figure 13.17. An array of structures.

A sample of the interaction and output is:

```
Reading a new stock item.
Enter the three dimensions of a board: 1 4 10
Enter the type of wood: walnut
Enter the price: $29.50
Enter the quantity in stock: 10
walnut      1" x 4" x 10 feet  $29.50  stock: 10
OK, sold 3
walnut      1" x 4" x 10 feet  $29.50  stock: 7
```

**Accessing a member of one structure in an array.** We also can access a single member of a structure element. Both a subscript and a member name must be used to access an individual part of one structure in an array of structures. Again, no parentheses are needed because of the precedence of the subscript and dot operators. Write the subscript first, because the overall object is an array. The result of applying the subscript is a structure to which the dot operator is applied. The final result is one member of the structure. For example, to print the `price` of the second piece of lumber in the `stock` array, we would write this:

```
printf( "Second stock item: %.2f\n", stock[1].price );
```

This statement's output is

```
Second stock item: $5.45
```

**Arrays of structures vs. parallel arrays.** In Section 10.3, we used a set of **parallel arrays** to implement a table. In this representation, each array represented one column of the table, and each array position represented one row. An array of structures can represent the same data: Each structure is a row of the table and the group of structure members with the same name represents one column.

A modern approach to representation would favor grouping all the columns into a structure and using an array of structures. This combines related data into a coherent whole that can be passed to functions as a single argument. The method is convenient and probably helps avoid errors. The argument in favor of using parallel arrays is that the code is simpler. The column's array name and a subscript are enough to access any property.

---

Type `bool` was defined in Figure 13.2. We use it here as the return type of a function that compares two structured variables for equality.

```
bool boards_equal( LumberT board1, LumberT board2 )
{
    return (strcmp(board1.wood, board2.wood) == 0) &&
           (board1.height == board2.height) &&
           (board1.width == board2.width) &&
           (board1.length == board2.length) &&
           (board1.price == board2.price) &&
           (board1.quantity == board2.quantity);
}
```

---

**Figure 13.18. Comparing two structures in C.**

In contrast, with an array of structures, the array name, a subscript, and a member name are needed.<sup>8</sup> The other factor is whether the functions in the program process mainly rows or columns of data. Row processing functions favor the structure, while column processing functions are most practical with the parallel arrays. A mixed strategy also may be used: We might define an array of menu items for use with `menu_i()` or `menu_c()`, but group all other data about each menu choice into a structure and have an array of these structures parallel to the menu array.

The output from `print_inventory` is

```
spruce 2" x 4" x 12 feet long -> 27 in stock at $8.20
pine 2" x 3" x 10 feet long -> 11 in stock at $5.45
pine 2" x 3" x 8 feet long -> 35 in stock at $4.20
white oak 1" x 6" x 8 feet long -> 158 in stock at $5.80
On sale: walnut 1" x 3" x 6 feet long -> 300 in stock at $29.50
```

### 13.3.4 Comparing Two Structures

Unfortunately, there is one useful operation that no C compiler supports: **comparing two structures** in one operation<sup>9</sup>. For example, if we wish to know whether `sale` and `plank` have the same values in corresponding members, we cannot write this:

---

<sup>8</sup>Interestingly, a C++ class provides the best of both worlds; an array of class objects is a coherent object itself, but class functions can refer to individual members of the object simply, without using a dot or arrow. Further examination of this topic is beyond the scope of this book.

<sup>9</sup>This operation is not supported because some structures contain internal padding bytes, and the contents of those bytes are not predictable. The C standards committee chose to omit comparisons altogether rather than support a universally defined component-by-component comparison operator.

---

The printing loop stops at `n`, the number of actual entries in the table. It does not print empty array slots.

```
void print_inventory( LumberT stock[], int n, LumberT onsale ) {
    printf( "\n Our inventory of wood products:\n" );
    for (int k = 0; k < n; ++k) {      // Process the filled slots of array.
        if (equal_lumber( onsale, stock[k] )) printf( " On sale: " );
        else printf( "          " );
        print_lumber( stock[k] );
    }   printf( "\n" );
}
```

---

**Figure 13.19. The `print_inventory` function.**

---

These declarations are in the file "lumber.h". They are used by the program in Figure 13.21. The type declaration was discussed in Figure 13.9.

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

#define MAX_STOCK 5
#define MAX_NAME 16

typedef struct LUMBER {           // Type definition for a piece of lumber
    char wood [MAX_NAME];         // Variety of wood.
    short int height;             // In inches.
    short int width;              // In inches.
    short int length;             // In feet.
    float price;                 // Per board, in dollars.
    int quantity;                // Number of items in stock.
} LumberT;

LumberT read_lumber( void );
void print_lumber( LumberT b );
void sell_lumber( LumberT* board, int sold );           // Modifies the structure.
bool equal_lumber ( LumberT board1, LumberT board2 );
void print_inventory( LumberT stock[], int len, LumberT onsale );
```

---

**Figure 13.20.** Declarations for the lumber program.

```
if (sale == plank) ...// Not meaningful in C.
```

Instead, to find out whether two structured variables are equal, each member must be checked individually. A function that does such a comparison is shown in Figure 13.18.

In the `boards_equal()` function, we compare two structured variables of type `LumberT` and return the value `true` or `false`, depending on whether the structures are equal or unequal, respectively. We use a series of `&&` operations to make this comparison. We compare each pair of members using the appropriate comparison operator for that type. Comparing members may be as simple as using `==`; more complicated, as in calling `strcmp()`; or complex, as calling another function to compare two structured components. As long as each successive pair matches, we continue testing. If any pair is unequal, the run-time system will skip the rest of the comparisons and return false.

**Calling the comparison function.** In the `print_inventory` function, we use a loop to process an entire array of structures, comparing each item to a sale item, and printing.

### 13.3.5 Putting the Pieces Together

Throughout the previous sections, many declarations, statements, and functions have been discussed. Most of these have been gathered together into a single program. The declarations from the top of the program are shown in Figure 13.20; it contains the `LumberT` structure definition and the function prototypes. The main program is found in Figure 13.21. The function definitions are in Figures 13.14, 13.15, 13.16, 13.18, and 13.19. Each block of code in the program has a comment noting the section in which it was discussed; we do not duplicate any of that discussion here. A full run of the program generates the following output sequence:

```
Demo program for structure operations in C.

Our inventory of wood products:
spruce 2" x 4" x 12 feet long -> 27 in stock at $8.20
pine 2" x 3" x 10 feet long -> 11 in stock at $5.45
pine 2" x 3" x 8 feet long -> 35 in stock at $4.20
white oak 1" x 6" x 8 feet long -> 158 in stock at $5.80

white oak price is $5.80
white oak in stock: 158
Error: can't sell 200 white oak boards; have only 158.
```

This program incorporates the declarations and code fragments used to demonstrate structure declarations and operations in Sections 13.2 and 13.3. The included file "lumber.h" is given in Figure 13.20. Function definitions are in Figures 13.14, 13.15, 13.16, and 13.18.

```

int main( void )
{
    puts( "\n Demo program for structure operations.\n" );
    // Structured object declarations: Figures 13.10, 13.11, and 13.17.
    LumberT onSale;          // Uninitialized
    LumberT* p;              // Uninitialized
    LumberT plank = { "white oak", 1, 6, 8, 5.80, 158 };
    LumberT stock[5] = { { "spruce", 2, 4, 12, 8.20, 27},
                        { "pine", 2, 3, 10, 5.45, 11},
                        { "pine", 2, 3, 8, 4.20, 35 },
                        { "" }, // Remaining members will be set to 0.
                     };
    int n = 3;               // The number of items in the stock array.

    // Access parts of a structure: Figures 13.11, 13.12.
    stock[n++] = plank;      // Copy into the array.
    print_inventory( stock, n, onSale ); // Nothing is on sale.
    printf( "%s price is $%.2f\n", plank.wood, plank.price );
    p = &plank;             // p points at plank.
    printf( "%s in stock: %i\n", p->wood, p->quantity );
    sell_lumber( p, 200 );   // A pointer argument.

    // Function calls using structs: Section 13.3.2.
    p = &stock[n];          // Point to a struct in the array.
    *p = read_lumber();      // Store a struct return value.
    n++;                   // Keep the item counter current.
    print_lumber( *p );     // Echo-print stock[3].
    print_inventory( stock, n, *p ); // stock[3] is on sale.
    sell_lumber( p, 200 );   // Call by address, value.

    // Accessing an array of structures: Section 13.3.3.
    sale = stock[2];         // Copy a structure.
    stock[3] = read_lumber(); // Input into an array slot.
    print_lumber( stock[3] ); // Print one array element.
    sell_lumber( 3, &stock[3] ); // Call by address.
    printf( "\n Second stock item: $%.2f\n\n", stock[1].price );
    print_inventory( stock, n, onSale ); // Process array of structs.

}

```

Figure 13.21. Structure operations: the whole program in C.

```

Reading a new stock item. Enter the 3 dimensions of a board: 1 3 6
Enter the kind of wood: walnut
Enter the price: $29.50
Enter the quantity in stock: 300
walnut 1" x 3" x 6 feet long -> 300 in stock at $29.50

Our inventory of wood products:
spruce 2" x 4" x 12 feet long -> 27 in stock at $8.20
pine 2" x 3" x 10 feet long -> 11 in stock at $5.45
pine 2" x 3" x 8 feet long -> 35 in stock at $4.20
white oak 1" x 6" x 8 feet long -> 158 in stock at $5.80
On sale: walnut 1" x 3" x 6 feet long -> 300 in stock at $29.50

OK, sold 200 walnut
walnut 1" x 3" x 6 feet long -> 100 in stock at $29.50

Our inventory of wood products:
spruce 2" x 4" x 12 feet long -> 27 in stock at $8.20
pine 2" x 3" x 10 feet long -> 11 in stock at $5.45
pine 2" x 3" x 8 feet long -> 35 in stock at $4.20
white oak 1" x 6" x 8 feet long -> 158 in stock at $5.80
walnut 1" x 3" x 6 feet long -> 100 in stock at $29.50

```

## 13.4 Structures and Classes in C++

Structures give the programmer a way to collect related variables into a single object. This is a huge leap forward in our ability to represent data. The C++ class is another leap beyond that: it allows us to group functions with the data objects that they work on. Members of a class can be data, as in a C struct, or functions that operate on the class data members. Putting functions inside the class declaration organizes the code into meaningful modules and simplifies a lot of the syntax for using the data members. This chapter gives a brief introduction to the most important aspects of classes in C++.

Classes also support a high level of data protection, using the keywords `const` and `private`. Normally, class data members are declared `private`. Class functions can see and change private data members, but functions outside the class cannot. When objects are passed as parameters, the `const` keyword prevents modification by the function that was called. This gives the programmer control over data objects so that all changes to them must be made by the functions in the object's class. Functions are usually `public` so that they can be seen and used by all other parts of the program.

C++ also supports `struct`, but it is not often used<sup>10</sup>.

### 13.4.1 Definitions and Conventions.

- The word *class* is used for type declarations, very much like “struct” in C.
- An *object* is an *instance* of a class. We declare
- Class names normally start with an upper case letter. Object and function names start with lower case. Both use camelCase if the identifier is multiple words.

For example, suppose you have a class named *Student*. When a *Student* is created, it has all the parts defined by the class *Student* and we say it is an *instance* of *Student*. You might then have instances named *Ann*, *Bob*, *Dylan*, and *Tyler*. We can say that these people belong to the class *Student*, or have the type *Student*.

**Common Uses for a Class.** There are many kinds of classes. The most important are:

- A *data class* models a real-world object such as a ticket or a book or a person or a pile of lumber. We say it stores the *state* of that object.
- A *container*, or *collection class*, stores a set or series of data objects.
- A *controller class* is the heart of an application. It implements the logic of the business or the game that is being built. It often has a container object as a data member. One or more data objects are either in the container or in the controller class itself.

---

<sup>10</sup>This allows a C++ compiler to compile C code without changes. A C++ `struct` actually follows the same syntax and rules as a C++ `class` except for one detail: `struct` members default to public visibility, while `class` members default to private.

There is no rule about what should and should not be a data member of a class. Generally, the data members you define are those your code needs to store the data, do calculations, manage the files, and make reports.

#### Examples of data classes:

- TheaterTicket: Data members would be the name of a show, a location, a date, a list price, a seat number, and a sold/available flag.
- Book: The title, author, publisher, ISBN number, and copyright date.
- Lumber: the type of wood, dimensions, price per board, and the quantity in stock.

#### 13.4.2 Function Members of a Class

A class supplies the *context* in which all its members are viewed and all its functions are executed. We see this context in several ways:

- The full name of a function starts with the name of its class. For example, the full name of the print function in the LumberT class is: `LumberT::print()`. We could also define a function named print in all the other classes in the application. The compiler will not get them confused, because they are called using the name of an object, and the compiler chooses the right print function for that context.
- Suppose a Pet class has a `feed()` function two instances named tweety and spot. You would feed the pets like this: `tweety.feed(); spot.feed();` The compiler looks up the spot's type, sees that spot is a Pet, and selects the `feed()` function for Pets.
- When the `spot.feed()` executes, spot defines the context in which feeding happens. Spot's needs get met; tweety's needs are not relevant to feeding spot. We say that `spot` is the implied parameter to the `feed()` function.
- Some functions are static and are actually called using the class name. For example, suppose a Game class supplies a function that gives instructions for new players. Then it would be called thus: `Game::instructions()`

The functions in a class define the way that class can be used. A class function can “see” and change the data members. Normally, functions are defined to do the routine things that every type needs: initialization and I/O. Class functions fall into these general categories:

- *Constructors and destructors.* A constructor initializes a newly-created class instance. A destructor frees dynamic memory that is part of the instance, if any exists.
- *I/O functions.* An output function is responsible for formatting and printing the data for one object to a file or to the screen. An input function is responsible for reading the data for one item from a file or from the keyboard. Every class should provide an output function that outputs all of its data members. These are called `print()` and are almost essential for debugging.
- *Calculation functions.* These do useful work. They are highly varied and depend on the application. They might compute a formula, do text processing, or sort an array.
- *Accessor functions.* Accessors allow limited access by any outside class to the private members of current class.

**Constructors** A constructor has the same name as its class, and a class often has multiple constructors, each with a different set of parameters. Every time the class is instantiated, one of the constructors is called to initialize it.

Typically, a programmer provides a *constructor with parameters* that moves information from each parameter into the corresponding data member. It also initializes other data members so that the object is internally consistent and ready to use<sup>11</sup>.

Many classes also need a *default constructor* that fully initializes the object to default values, often 0's. A default constructor is necessary to make an array of class objects. If you do not provide a constructor for the class, the compiler will provide a *null default constructor* for you. A default constructor requires no parameters. A null constructor does no initializations.

---

<sup>11</sup>More advanced programmers sometimes provide copy constructors and move constructors.

**Destructors** There is only one destructor in a class. It is run each time a class instance goes out of scope and “dies”. This happens automatically whenever control leaves the block of code in which the object was instantiated. Many programs create and use a temporary object inside a block of code. At the end of the block, the object’s destructor is called and the object is deallocated.

If part of an object was dynamically allocated, the programmer is responsible for managing the dynamic memory. The destructor of the class that includes the dynamic part should call `delete`, which calls the destructor, which causes the memory to be freed.

**Accessors.** There are two kinds of accessors: getters and setters. A getter function gives a caller read-only access otherwise-private data, and is often given a name such as `getCount()` or `getName()`. A data class will often have one or more getters. However, it is rarely appropriate to provide a getter for every data member. Doing so is evidence of poor OO design: most things that are done with data members should be done by functions in the same class.

Setters allow an outside function to modify the private parts of a class instance. This is rarely necessary and can usually be avoided by proper program design. There is no reason to provide a setter for every data member and doing so is just sloppy programming.

### 13.4.3 Encapsulation

One of the most important properties of a C++ class is that it can *encapsulate* data. Each class has a responsibility to ensure that the data stored in its objects is internally consistent and meaningful at all times:

- Declare all data members to be private so that only functions from the same class can modify the data members.
- Use the constructor to get the data into a new object.
- Make sure that every function in the class maintains consistency. The value stored in each data member must make sense in the context of the other data members.

Any class member can be defined to be either public or private. Public members are like global variables – any function anywhere can change them. Private members are visible only to functions in the same class<sup>12</sup>.

Design principle 1: A class protects its members.

When the data members are private, the class is able to control access to them. It also has the responsibility of providing public functions to carry out all the important actions that relate to the class: validation, calculation, printing, restricted access, and maybe even sorting.

Design principle 2: A class should be the expert on its own members.

This is why a well designed OO program does not need getters and setters: almost all the work is done by a series of smaller and smaller classes.

**Delegation** In an OO program, much or most of the work is done by *delegation*. The main program creates an object, *A* then calls functions that belong to *A*’s class to get the work done. *A* often has smaller objects (*B* and *C*) within it, so *A*’s functions call functions in the classes of *B* and *C* to get the work done. Work orders get passed down the line from `main()` to all the other functions.

Design principle 3: Delegate the activity to the expert.

This is why a well designed OO program does not need getters and setters: almost all the work is done by a series of smaller and smaller classes.

---

<sup>12</sup>Later, you will also learn about protected variables.

### 13.4.4 Instantiation and Memory Management

Instantiation is the act of creating an object, that is, an instance of a class. This can be done by two methods:

- Declaration creates *automatic* storage. When you declare a variable of a class type, space is allocated for it on the run-time stack. That space will remain until control leaves the block of code in which the object was declared. At block-end time, the stack frame is deallocated and the destructors will be run on all objects in the frame. No explicit memory management is needed, or permitted, on these automatic variables.
- Dynamic allocation. At any time during execution, a program can call `new` to instantiate a class. The result is a pointer to an object. The program is then responsible for explicitly freeing this object when it is no longer needed. For example, assume we have a class named `Student`. Then the following code allocates space to store the `Student` data, calls the 3-parameter `Student` constructor, and stores the resulting pointer in `newStu`: `newStu = new Student("Ann", "Smith", 00256987);`

This dynamic object can be passed around from one scope to another and will persist until the program ends. It will probably be put into a container class. Suppose that later, when this data is no longer needed, it is attached to a pointer named `formerStu`. We deallocate it thus: `delete formerStu;`

Design principle 4: Creation and deletion. With dynamic allocation, new objects should be created by the class that will contain them, and that class is responsible for deleting them when they are no longer needed.

## 13.5 The Lumber Program in C++

This is the C++ analog of the C program in the previous section. The output is the same, and the code is parallel throughout. However, the C++ version is written to follow the design principles of encapsulation, expertise, delegation, and creation. The discussion below will focus on the differences between the C and C++ versions.

**Compile modules and files.** This program is organized into two modules: a main module (`main.cpp`) and a lumber class module (`lumber.hpp` and `lumber.cpp`). The compiler will translate each `.cpp` file separately, then link them together and also link with the standard libraries. This modular organization is used for all C and C++ programs except the very small ones.

Header files (`.hpp`) contain only class, enum, and `typedef` declarations. They do not contain variables or arrays. The functions are prototyped within the class declaration, and normally defined in a separate code file (`.cpp`). This type information is needed at compile time to translate function calls, so any module that calls a function in class *A* must `#include` the header file for class *A*. One `#include` statement is written at the top of each `.cpp` file.

The major difference between C and C++ code is the way functions are called. A C function has all of its parameters inside the parentheses that follow the function name. In contrast, a C++ has an *implied parameter* whose name is written *before* the function name. The rest of the parameters are written between the parentheses.

### Notes on Figure 13.22: The header file for the LumberT class.

- Note that the file name is written in the upper-right corner of each file. Please adopt this very useful habit.
- The data members are the same as the C version except that we are using a C++ string instead of a char array for the kind of wood. This makes input easier.
- The function prototypes are inside the class declaration (lines 15..24).
- This class has two constructors, one with parameters, one without (lines 15..16).
- A default constructor is prototyped and defined on line 16. The keyword `= default` makes it very convenient to initialize everything in the object to 0 bits.

```

1 // Header file for the lumber class                      file: lumber.hpp
2 #include "tools.hpp"
3 #define MAX_STOCK 5
4
5 class LumberT {           // Type definition for a piece of lumber
6     private:
7         string wood;          // Variety of wood.
8         short int height;    // In inches.
9         short int width;     // In inches.
10        short int length;   // In feet.
11        float price;         // Per board, in dollars.
12        int quantity;        // Number of items in stock.
13
14 public:
15     LumberT( const char* wo, int ht, int wd, int ln, float pr, int qt );
16     LumberT() = default;
17     void read();
18     void print() const;
19     void sell( int sold );           // Modifies the object.
20     bool equal( const LumberT board2 ) const;
21
22     string getWood() const { return wood; }
23     float getPrice() const { return price; }
24     int getQuantity() const { return quantity; }
25 };

```

---

**Figure 13.22. The LumberT Class in C++**

- Note that the function names here are shorter than in the C program. This is possible because the class name, `LumberT`, always is part of the context in which these functions are called.
- A class function is called using the name of a class instance. That class instance is the object that the function reads into or prints or compares. This will become clearer when `main()` is discussed.
- Lines 22...24 are accessor functions. Although there are six data members, there are only three getters. Getters are simply not needed for many class members.
- There are no setters at all. Data is put into objects by using the constructor with parameters or using assignment to copy one entire object into another.

**Notes on Figure 13.23: The lumber main program.** This Figure has the main function and a print function called by main.

- Line 30 includes the header for the tools module. You will see this in most C++ programs in this text. The tools header brings in all the other header files you are likely to need. The tools module supplies three functions that are extremely helpful:
  - `banner()` prints a heading on your output, including your name and the date of execution<sup>13</sup>.
  - `bye()` prints a termination comment that proves to your instructor that your program terminated normally.
  - `fatal()` prints an error comment, flushes the output buffers, closes your files, and aborts execution cleanly. You should call `fatal()` any time the program cannot continue to do its work. It is called with a format and an output list like `printf()`. Your format string should give clear information about the nature of the error.

---

<sup>13</sup>To make this work, you need to enter your own name once on line 2 of `tools.hpp`.

```

27 // -----
28 // A C++ program corresponding to the Lumber Demo program in C.    main.cpp
29 //
30 #include "tools.hpp"
31 #include "lumber.hpp"
32 void printInventory ( LumberT stock[], int n, LumberT& onSale );
33
34 //
35 int main( void ) {
36     cout <<"\n Demo program for structure operations in C++.\\n";
37
38     // Struct object declarations: -----
39     LumberT onSale;          // Initialize using the default constructor.
40     LumberT* p;              // Uninitialized
41     LumberT plank( "white oak", 1, 6, 8, 5.80, 158 ); // Call first constr.
42     LumberT stock[5] = { // Call first constructor three times.
43         { "spruce", 2, 4, 12, 8.20, 27},
44         { "pine",   2, 3, 10, 5.45, 11},
45         { "pine",   2, 3, 8, 4.20, 35 },
46         {} // Call default constructor.
47         // Call default constructor for additional elements.
48     };
49
50     int n = 3;                // The number of items in the stock array.
51     stock[n++] = plank;       // Copy into the array.
52     printInventory( stock, n, onSale ); // Nothing is on sale.
53
54     // Access parts of a structure -----
55     cout <<fixed <<setprecision(2) <<" "                         // Use object part.
56         <<plank.getWood() <<" is $" <<plank.getPrice() <<"\\n ";
57     p = &plank;                           // Point to object.
58     cout <<plank.getWood() <<" in stock: " <<p->getQuantity() <<endl;
59     p->sell( 200 );                     // Use the pointer.
60
61     // Use a struct object to call a struct member function. -----
62     p = &stock[n];                  // Point to an object.
63     p->read();                      // Read into stock[4].
64     p->print();                     // Echo-print stock[4].
65
66     printInventory( stock, 5, *p ); // stock[3] is on sale.
67     p->sell( 200 );                // This will modify the sale object.
68
69     // Test if two structures are equal; Section 13.3.4. -----
70     printInventory( stock, MAX_STOCK, onSale );
71 }
72
73 //
74 // This function is global, not part of the LumberT structure.
75 void printInventory ( LumberT stock[], int n, LumberT& onSale ){
76     cout <<"\n Our inventory of wood products:\\n";
77
78     for (int k = 0; k < n; ++k) {      // Process every slot of array.
79         if ( onSale.equal( stock[k] ) ) cout <<" On sale: ";
80         else cout <<"           ";
81         stock[k].print();
82     }
83     cout <<endl;
84 }
```

Figure 13.23. The Lumber program in C++

- Line 31 includes the header file for the lumber module. You must do this to be able to declare lumber objects and call the lumber functions.
- Line 32 is the prototype for the only function that does not belong in the lumber class.
- Lines 39–48 create the same lumber objects as in the C program.
- Lines 50–52 are exactly the same as the C version.
- Compare lines 55–56 to the `print_lumber()` function in C. C++ output is quite different from C.
  - `fixed` means approximately the same thing as `%f`.
  - To print an amount in dollars and cents, Use `fixed` and `setprecision(2)`.

Many people believe that C provides easier ways to control formatting, especially for types `float` and `double`.

- Lines 63 calls a class function using `p`, a pointer to a class object. In the C++ program, the object name is first, and the function reads info directly into the object `p` points at. This is a different mode of operation from the C program, which reads the data into a local temporary and returns a copy of that temporary that is finally stored in the object `p` points at. The C++ program is more direct and more efficient.
- On lines 64 and 67, again, the pointer `p` provides the context in which the `print()` and `sell()` functions operate, and the functions called are the ones defined for `p`'s object. In the C program, `p` is written as a parameter to the `print` function and the `sell` function.

**Notes on the `printInventory()` function.** The `print` function is not in the `Lumber` class because it involves an entire array of lumber.

Design principle 5: The functions that belong in a class are those that deal with one class instance. A function that handles many class instances should be in a larger class or in `main`.

- The first two parameters to this function are the array of lumber and the number of valid, initialized entries in that array. Every function that processes an array must stop processing at the end of the valid entries. These parameters are passed by value, that is, copies are made in the stack frame for the function.
- The third parameter is a `LumberT` object that is passed by reference (`&`), that is, the address of the object is passed to the function and the function uses main's object indirectly. Call by reference avoids the time and space expense of making a copy and is normally used for large objects. Within the function, the syntax for using this parameter is the same as the syntax used with call by value.
- Line 79 calls the `equal()` function in the `lumber` class because `onSale` is declared to be a `LumberT` object. The `equal()` function will compare two lumber objects: (a) `onsale`, and (b) one entry in the stock array.
- On the basis of the outcome, the `printInventory` function will either print some spaces or print the words “On sale:”, then the array entry will be printed.

#### Notes on Figure 13.24: The functions for the `LumberT` class.

- Within a class function, the name of a class member refers to that member of whatever object was used to call the function. That object is called the *implied parameter* and it forms the context in which the function executes.
- The constructor with parameters. Lines 85–92 define the primary class constructor. Note that a constructor does not have a return type. Each line takes the value of one parameter and stores it in the newly-created class instance. The parameter names here are short versions of the corresponding member name. That is OK.
- Another common style is to use the same name for the parameter and for the class member. In that case, the assignments would be written like this: `this->wood = wood`; “this” is a keyword. It is a pointer to the implied parameter, and `this->` can be used to select a part of the object you are initializing.

---

```

82 #include "lumber.hpp"                                // file: lumber.cpp
83
84 // Constructor: -----
85 LumberT::LumberT( const char* wo, int ht, int wd, int ln, float pr, int qt ) {
86     wood = wo;
87     height = ht;
88     width = wd;
89     length = ln;
90     price = pr;
91     quantity = qt;
92 }
93
94 // -----
95 // Read directly into the object -- no need for a temporary variable.
96 void LumberT::read() {
97     cout << " Reading a new item. Enter the 3 dimensions of a board: ";
98     cin >> height >> width >> length;
99     cout << " Enter the kind of wood: ";
100    cin >> ws;
101    getline( cin, wood );
102    cout << " Enter the price: $";
103    cin >> price;
104    cout << " Enter the quantity in stock: ";
105    cin >> quantity;
106 }
107
108 // -----
109 void LumberT::sell( int sold ){
110     if (quantity >= sold) {
111         quantity -= sold;
112         cout << " OK, sold " << sold << " " << wood << endl;
113         print();
114     }
115     else {
116         cout << " Error: can't sell " << sold << " boards; have only "
117             << quantity << "\n";
118     }
119 }
120
121 // -----
122 void LumberT::print() const {
123     cout << " " << wood << ": "
124         << height << " x " << width << ", " << length << " feet long -> "
125         << quantity << " in stock at $"
126         << fixed << setprecision(2) << price << endl;
127 }
128
129 // -----
130 bool LumberT::equal( const LumberT board2 ) const {
131     return (wood == board2.wood) &&
132         (height == board2.height) &&
133         (width == board2.width) &&
134         (length == board2.length) &&
135         (price == board2.price) &&
136         (quantity == board2.quantity);
137 }
```

---

Figure 13.24. The Lumber functions in C++

Use this diagram to understand the next program.

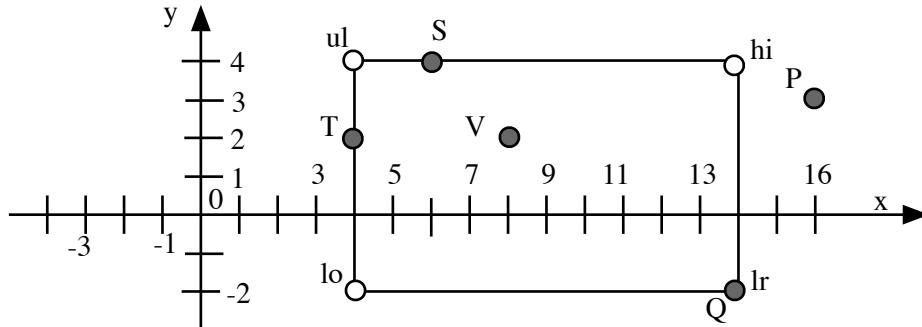


Figure 13.25. A rectangle on the  $xy$ -plane.

- The `read()` function. This is very much easier to read and write than the `scanf()` statements in the C code: no format codes, no ampersands, no fuss. Also, there is no local variable and no return statement. The data that is read is stored in the implied parameter and is returned that way.
- Note how easy it is to read input into a C++ string (lines 100-101). There is no need to worry about buffer overflow because strings automatically resize themselves, if needed. The `>>ws` on line 44 will eliminate the newline character on the end of the previous line plus any whitespace before the visible characters on the current line.
- The `sell()` function is parallel to the C version. Only the output statements have been changed. Notice that `print_lumber( *board )` in C is simplified to `print()` in C++. We don't need a parameter because the object used to call `sell()` replaces it. We don't need to write *anything* in front of the function name, because it is called from another function in the same class.
- The `print()` function is parallel to the C version, although the code looks quite different. In C++, class members can be used directly by any class function. So we write `height14` instead of using a parameter name, as in `b.height`.
- The `equal()` function is exactly like C version except for `board1.` written in front of each member name. The role played by `board1` is covered by the implied parameter in C++

## 13.6 Application: Points in a Rectangle

The next —pl C program example uses a structure to represent a point on the  $xy$  plane. It illustrates a variety of operations on structures, including assignment, arithmetic, and function calls.

An elementary part of any CAD (computer-aided design) program or graphics windowing package is a function that determines whether a given point in the  $xy$  plane falls within a specified rectangular region. In a mouse-driven interface, the mouse is used to select the active window, “grab” scrolling buttons, and “click on” action buttons and menu items. All these rectangular window elements on the screen are represented inside the computer by rectangles, and the mouse cursor is represented by a point. So, quite constantly, a mouse-based system must answer the question, “Is the point  $P$  inside the rectangle  $R$ ?”

We present a program that answers this simple but fundamental question. The problem specifications are given in Figures 13.25 and 13.26. Figure 13.27 outlines a test plan. The various portions of the program are contained in Figures 13.30 and 13.31 through 13.34 and Figure 13.28 diagrams the structured data types used in this application. Structures are used to define both points and rectangles in this program. A point is represented by a pair of doubles (its  $x$  and  $y$  coordinates) and a rectangle is represented by a pair of points (its diagonally opposite corners). We use the first structured type (`point`) to help define the next. This is typical of the way in which complex **hierarchical data structures** can be constructed by combining simpler parts.

<sup>14</sup>We could write `this->height`, but there is no reason to write `this->`. The extra words are just clutter.

---

Refer to the diagram in Figure 13.25.

**Problem scope:** Given the coordinates of a rectangle in the integer  $xy$  plane and the coordinates of a series of points, determine how the position of each point relates to the rectangle. As an example, in the diagram in Figure 13.25,  $R$  has diagonally opposite corners at  $lo = (4, -2)$  and  $hi = (14, 4)$ . Point  $P = (16, 3)$  is outside the rectangle,  $Q = (14, -2)$  is at a corner,  $S = (6, 4)$ , and  $T = (4, 2)$  are on two of the sides, and  $V = (8, 2)$  is inside the rectangle.

**Limitations:** The sides of the rectangle will be parallel to the  $x$  and  $y$  axes. Thus, two diagonally opposite corners are enough to completely specify the position of the rectangle.

**Input:** In Phase 1, the user enters the  $x$  and  $y$  coordinates (real values) of points  $lo$  and  $hi$ , the lower left and upper right corners of the rectangle. In Phase 2, the user enters the  $x$  and  $y$  coordinates of a series of points. An input point at the same position as  $lo$  will terminate execution.

**Output:** In Phase 1, echo the actual coordinates of the corners of the rectangle that will be used. In Phase 2, for each point entered, state whether the point is outside, at a corner of, on a side of, or inside the rectangle.

**Formulas:** A point is outside the rectangle if either of its coordinates is less than the corresponding coordinate of  $lo$  or greater than the corresponding coordinate of  $hi$ . It is inside the rectangle if its  $x$  coordinate lies between the  $x$  coordinates of  $lo$  and  $hi$  and the point's  $y$  coordinate lies between the  $y$  coordinates of  $lo$  and  $hi$ . The point is on a corner if both of its coordinates match the  $x$  and  $y$  coordinates of one of the four corners. It is on a side if only one of its coordinates equals the corresponding  $x$  or  $y$  coordinate of a corner and the point is not outside nor on a corner.

**Computational requirements:** If the user accidentally enters the wrong corners of the rectangle or enters the corners in the wrong order, the program should attempt to correct the error. The program also must function sensibly if the rectangle is degenerate; that is, if the  $x$  coordinates of  $lo$  and  $hi$  are the same, the  $y$  coordinates of  $lo$  and  $hi$  are the same, or if  $lo$  and  $hi$  are the same point. The first two cases define a line; the last defines a point. If rectangle  $R$  is a single point, then point  $P$  must be either “on a corner” or “outside”  $R$ . If the rectangle is a straight line,  $P$  can be “on a corner,” “on a side,” or “outside”  $R$ .

---

**Figure 13.26. Problem specifications: Points in a rectangle.**

Just as some programs take simple objects as input and operate only on those simple objects (`floats` or `ints`), conceptually, this program takes structured objects as data and operates on these structured objects. Because operations on compound objects are more complex than those on simple objects, functions are introduced to perform these operations. This keeps the main flow of logic simple and lets us focus separately on the major operations and the details of the structured objects. New function prototypes are introduced in which the `struct` and `enum` data types are used to declare both parameters and return types. Specifically, we use the enumerated type `in_type` declared in Figure 13.2, which has one code for each way that a point's position can relate to a rectangle.

**A test plan for points in a rectangle.** The specifications in Figure 13.26 ask us to analyze the position of a point relative to a rectangle. They require us to handle any point in the  $xy$  plane and any rectangle with sides parallel to the  $x$  and  $y$  axes. Degenerate rectangles (vertical lines, horizontal lines, and points) must be handled appropriately. A suitable test plan, then, will include at least one example of each case.

We construct such a plan, shown in Figure 13.27 by starting with the sample rectangle and points given in Figure 13.25. We then add points that lie on the other two vertical and horizontal sides. The last input for this rectangle is a point equal to the lower-left corner, which should end the processing of this rectangle. This part of the test plan is shown in the top portion of Figure 13.27. Next, we add two rectangles whose corners have been entered incorrectly, to ensure that the program will swap the coordinates properly and create a legitimate rectangle. Finally, we need to test three degenerate rectangles: a vertical line, a horizontal line, and a point.

Type of Rectangle	Lower Left ( $x, y$ )	Upper Right ( $x, y$ )	Point ( $x, y$ )	Position (answer)
Normal	(4, -2)	(14, 4)	$P = (16, 3)$ $Q = (14, -2)$ $S = (6, 4)$ $(5, -2)$ $T = (4, 2)$ $(14, -1)$ $V = (8, 2)$ $lo = (4, -2)$	Outside Corner Horizontal side Horizontal side Vertical side Vertical side Inside Corner, end of test
$x$ coordinates, reversed $x$ coordinates, corrected	(2, 0) (-1, 0)	(-1, 5) (2, 5)	(1, 1) (-1, 0)	Inside Corner, end of test
$y$ coordinates, reversed $y$ coordinates, corrected	(-1, 5) (-1, 0)	(2, 0) (2, 5)	(-1, 6) (-1, 0)	Outside Corner, end of test
Vertical line	(1, 0)	(1, 3)	(1, 3) (1, 2) (1, 0)	Corner Side Corner, end of test
Horizontal line	(0, 1)	(3, 1)	(3, 1) (2, 1) (2, 0) (0, 1)	Corner Side Outside Corner, end of test
Point	(1, 3)	(1, 3)	(2, 0.0001) (1, 3)	Outside Corner, end of test

Figure 13.27. The test plan for points in a rectangle.

### 13.6.1 The program: Points in a Rectangle

Figure 13.29 contains the type declarations and function prototypes for the program, and is stored in the file named `rect.h`. Corresponding function definitions are found in the file `rect.c` and in Figures 13.31 through 13.34. The `main()` program is in Figure 13.30 and the file `main.c`; it prints program titles and executes the usual work loop. The test plan is given in Figure 13.27 and diagrams of the structured types are in Figure 13.28.

Notes on Figure 13.29: Header file for points in a rectangle.

*First box: type declarations.*

- A header file typically begins with all the include commands that will be needed by function in this module.
- Following that are the type declarations. This placement permits all the functions prototyped below to refer to the newly defined types.

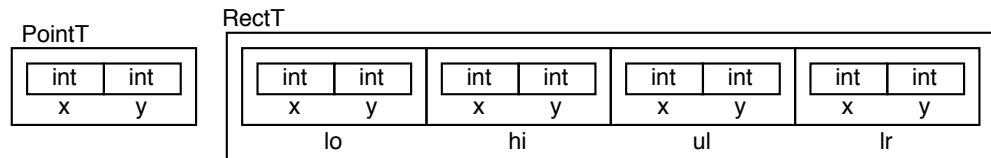


Figure 13.28. Two structured types.

This part of the program declares the necessary type definitions and function prototypes. It must be included in main.c and in rect.c.

```
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h> // for tolower()

typedef struct POINT { int x, y; } PointT ;
typedef struct RECTANGLE { PointT lo, hi, ul, lr; } RectT;
typedef enum INTYPE { P_IN, P_SIDE, P_CORNER, P_OUT } InType;

bool equal( PointT p1, PointT p2 );
bool over ( PointT p1, PointT p2 );
bool under( PointT p1, PointT p2 );
bool left ( PointT p1, PointT p2 );
bool right( PointT p1, PointT p2 );

void swap( int * p1, int * p2 );
RectT makeRect( int x1, int y1, int x2, int y2 );
void testPoints( RectT r );
InType locate( RectT r, PointT p );
```

Figure 13.29. Header file for points in a rectangle.

- The type `PointT` represents the coordinates of a point in the integer  $xy$  plane. We define it as a `struct` containing a pair of integers, `x` and `y`. Although we could define this type as an array of two integers, using a structure provides a clearer and more natural way to refer to the parts. By using a *structure*, we can name the parts, not number them. This type is diagrammed on the left in Figure 13.28.
- According to the problem specifications, a rectangle is defined by the two points at its opposite corners, called `lo` and `hi`. The type `RectT`, therefore, has two `PointT` data members named `lo` (the lower-left corner) and `hi` (the upper-right corner). In addition, there are data members for the two remaining corners of the rectangle. Having these as class members is not necessary, but it makes it easy to test whether an input point is on a corner. The compound type is diagrammed on the right in Figure 13.28. We could have used an array of integers but that does not capture the fundamental nature of the problem. We prefer a structure of structures because that lets us refer to each part by a name that makes sense to a reader.
- To name the members of this structure, we choose brief but suggestive names so that the code will not be too wordy. For example, to access the `x` coordinate of the lower-left corner of rectangle `r`, we write `r.lo.x`.
- The third `typedef` declares `InType`, an enumeration of the positions a point can have relative to a rectangle. Each time a point is entered, the function `locate()` is called to test its position, and one of these codes is returned as the result of the function. These codes permit us to return the position information from a function with great ease and clarity. The order in which the codes are listed here does not matter.

#### *Second box: the prototypes for PointT*

- The first prototype is for the `equal()` function in Figure 13.33. It has two `PointT` parameters and compares each member of the first parameter to the corresponding member of the other. If both pairs are equal, the points are equal, and the return value is `true`. If either pair fails to match, `false` is returned.
- The next four prototypes compare one member of `p1` to the corresponding member of `p2`. The names `over`, `under`, `left`, and `right` were chosen to make sense: `p1` can be over `p2`, under it, to its left, or to its right. These functions allow us to write brief, clear code to test the position of a point relative to a rectangle.
- The last prototype in this box is a print function. Every structure should have a print function. Sometimes it is needed for output, but it is always needed for debugging. A print function formats the data members of the class in a readable format.

#### *Third box: the prototypes for RectT*

- The first prototype is for `swap()`, defined in Figure 11.10. Swap needs no explanation at this point.

```

#include "rect.h"

// Main program for points in a rectangle ----- file: main.c
int main( void ){
    char again;           // For user response in testPoints loop.
    int xLo, yLo, xHi, yHi; // To input the coordinates of the rectangle.

    puts( "\n Point in a Rectangle Program\n"
          " Given a rectangle with a side parallel to the x axis \n"
          " and a series of points on the integer xy plane,\n"
          " say where each point lies in relation to the rectangle." );

    do{
        printf( "\n Enter x and y for lower left corner:  " );
        scanf( "%i%i", &xLo, &yLo );
        printf( " Enter x and y for upper right corner:  " );
        scanf( "%i%i", &xHi, &yHi );

        RectT rect = makeRect( xLo, yLo, xHi, yHi ); // Create the rectangle.
        testPoints( rect );

        printf( "\n Do you want to test another rectangle (y/n)?  " );
        scanf( " %c", &again );
    } while (tolower( again ) != 'n');

    return 0;
}

```

**Figure 13.30.** Main program for points in a rectangle.

- The second prototype is for the function `makeRect()` in Figure 13.31, which reads and validates the coordinates of a rectangle, then returns the rectangle as a structure. This function will be replaced by a constructor in the C++ version.
- The third prototype is for the `testPoints()` function in Figure 13.32. It reads in a series of points, locates those points relative to the rectangle, and prints the results.
- The fourth prototype is for `locate`, which is defined in Figure 13.34. The parameters are a rectangle and a point. The result of the function is an `InType` value; this means that the function will return one of the constants from the `InType` enumeration.
- The last prototype in this box is a print function. Every structure should have a print function for output and for debugging.

#### Notes on Figure 13.30: Main function for points in a rectangle.

- First box. Following that is an include command that brings in the header information (type definitions and prototypes) for the type `RectT`, used in the third box.
- This main program is in a file that is separate from the other parts of the application. The comment line identifies the purpose and the name of that file.
- Second box. A main function typically prints an identifying heading Then it creates a data object (third box) and uses that data object to do the job. If files need to be opened, main often opens them. If a query loop is needed, it is in main.
- Third box. This main program contains a query loop that processes a series of rectangles. The query loop permits this to continue as long as the user wishes.
- First inner box. Each time through the loop, the program prompts for and reads the two defining corners of a rectangle. The program prompts for the rectangle's two corners one at a time. It would be possible to

Construct a structured object and initialize its members to the parameter values. File: `rect.c`.

```
// Read, validate, and return the coordinates of a rectangle.
RectT
makeRect( int xLo, int yLo, int xHi, int yHi ){
    RectT r;
    // If corners were not entered correctly, swap coordinates.
    if (xLo > xHi) swap( &xLo, &xHi );
    if (yLo > yHi) swap( &yLo, &yHi );
    printf( "\n Using lower left = (%i, %i), upper right = (%i, %i).\n",
            xLo, yLo, xHi, yHi );

    // Set coordinates of all four corners.
    r.lo.x = r.ul.x = xLo;                      // Left side of rectangle.
    r.lo.y = r.lr.y = yLo;                      // Bottom of rectangle.
    r.hi.x = r.lr.x = xHi;                      // Right side of rectangle.
    r.hi.y = r.ul.y = yHi;                      // Top of rectangle.

    return r;
}
```

**Figure 13.31. Making a rectangle.**

read both corners (all four coordinates) in one statement, but entering four inputs at one prompt is likely to cause confusion or omissions.

- Second inner box. Once the coordinates have been read in, it is possible to make a rectangle. The `makeRect` function stores the coordinates in the object it creates and returns. Then the newly-initialized object is sent to `testPoints()`, which inputs a series of points and locates them relative to the rectangle.
- Note that the main program is not involved with any of the details of the representation or processing of rectangles and points. All that **work is delegated** to other functions, and each of them accomplishes just one part of the overall job. This kind of modular program construction is the foundation of modern programming practice.

**Notes on Figure 13.31: Making a rectangle.** This function illustrates input, validation, and returning a structure. The coordinates were read in `main()` and are passed as parameters to this function.

***First box: testing for valid input.***

- The correct operation of the algorithm depends on having the coordinates of the lower-left corner be less than or equal to the coordinates of the upper-right corner. The user is instructed to enter the points in this order. However, trusting a user to obey instructions of this sort is never a good idea. A program that depends on some relationship between data items should test for that relationship and ensure that the data are correct.
- If the user enters the wrong corners (upper left and lower right) or enters the correct corners in the wrong order, the program still has the information needed to continue the process. That information must be stored in the correct members of the rectangle structure. So we test for this kind of an error and swap the coordinates if necessary.
- After initializing the corners of the rectangle properly, the points `lo` and `hi` are printed, using the usual mathematical notation. ( $x$ ,  $y$ ).
- The program prints the `lo` and `hi` points after testing for errors because coordinates might have been swapped. A program always should echo its input so that the user knows what actually is being processed. This is essential information during debugging, when the programmer is trying to track down the cause of

---

Input and test a series of points.

File: rect.c.

```
const char* answer[4] = {"inside", "on a side of", "at a corner of", "outside"};
```

```
void testPoints( RectT r ) {
    PointT pt;                                // The points we will test.
    InType where;                             // Position of point relative to rectangle.
    puts( " Please enter a series of points when prompted." );
    do {
        printf( "\n Enter x and y (%i %i to quit): ", r.lo.x, r.lo.y );
        scanf( "%i%i", &pt.x, &pt.y );
        where = locate( r, pt );
        printf( " The point (%i, %i) is %s the rectangle.\n",
            pt.x, pt.y, answer[where] );
    } while ( !equal( pt, r.lo ) );
}
```

---

**Figure 13.32. Testing points.**

an observed error. Being certain about the data used reduces the number of factors that must be considered. Echoing the input also gives the user confidence that the program is progressing correctly.

- As an example, if two points were entered incorrectly, the user might see the following dialog. Note that the *y* coordinates get corrected:

```
Enter x and y for lower left corner: -1 5
Enter x and y for upper right corner: 2 0
```

Using lower left = (-1, 0), upper right = (2, 5).

**Second box: setting the rectangle's coordinates.**

- Each of the four coordinates is shared by two corners of the rectangle. This is made clear by assigning each value to two points.
- The assignment statements show why we choose short names for the parts of the structures, instead of longer names like `upperRight` and `xCoordinate`. Since we may have to write a long list of qualifying member names to specify an individual component, even short names can result in long phrases.

Design principle #6: Keep it short and simple.

If member names are long, the statements are difficult to write on one line and they become hard to read and error prone. It is much easier to read a brief expression like the first version that follows than a long-winded version like the second:

```
r.lo.x = r.ul.x = xLo;
rectangle.lowerLeft.xCoordinate = rectangle.upperLeft.xCoordinate = xLo;
```

**Third box: returning the rectangle.** A `RectT` variable `r`, is declared at the top of the `makeRect()` function. It is a local variable in this function and will be deallocated when the function returns. Within the function, `r`'s members are initialized to the parameter values. When control reaches the `return` statement, `r` contains eight integers representing the four corners of a rectangle.

The `return` statement sends a copy of this structured value containing these four points back to `main`, then immediately deallocates the original. This structured value must be stored in a `RectT` variable in `main`.

**Notes on Figure 13.32. Testing points.** We illustrate some basic techniques for using enumerations and structures: variable declarations, output, and function calls that use and return structured data.

---

Here are five one-line functions to compare two points.

```
bool equal( PointT p1, PointT p2 ){ return p1.x == p2.x && p1.y == p2.y; }
bool over ( PointT p1, PointT p2 ){ return p1.x > p2.x; }
bool under( PointT p1, PointT p2 ){ return p1.x < p2.x; }
bool left ( PointT p1, PointT p2 ){ return p1.y < p2.y; }
bool right( PointT p1, PointT p2 ){ return p1.y > p2.y; }
```

---

**Figure 13.33. Comparing two points.**

**First box: An array of strings.** To allow convenient and readable output of the results, we declare an array of strings parallel to the enumeration constants. Each string in this array is written in plain English and describes the location of a point w.r.t. the given rectangle. This array is used in the inner box.

**Second box: processing points.**

- We use a `do...while` sentinel loop to process a series of points. On entering the loop, the program prompts for and reads a point, explaining clearly how to end the loop (by entering the point `1o` as a sentinel). When reading data into a structured variable (or printing from it), it is necessary to read or print each member of the structure separately. The program cannot scan data directly into `pt` as a whole but must scan into `pt.x` and `pt.y`. Similarly, it cannot print the lower-left corner of `r` by printing `r.lo`; it must print the individual members, `r.lo.x` and `r.lo.y`.
- In the inner box, the program calls the function `locate()` to determine the position of point `pt` relative to rectangle `r`. The two arguments are structured values. A structured argument is passed to a function in the same manner as an argument of a simple type, that is, by copying all of the structure's members. This function returns an `InType` value, which is stored in `where`, an `InType` variable.
- Since enumerated types are represented by small integers, the program could print the `InType` value directly, in an integer format. However, this would not be very informative to the user. Instead, we use the integer as a subscript to select the proper phrase from the parallel array of answers defined in the first box. The `printf()` statement prints the resulting string using a `%s` format.
- At the end of the loop, the program calls the `equal()` function to determine whether the point `pt` is the **sentinel value**; that is, the same point as the lower-left corner of the rectangle. If so, we leave the loop. The `equal()` function returns a `true` or `false` answer, which can be tested directly in a `while` or `if` statement. We simplify and clarify the logic of `testPoints()` considerably by putting the testing details into the `equal()` function.

**Notes on Figure 13.33: Comparing two points** These five functions express positions on the plane in intuitive language that relates directly to positions on the plane. Creating a function for something that can be written in one line may seem unusual. However, factoring out this detail makes the `locate()` function in Figure 13.34 much cleaner and easier to understand. This, in turn, leads to arriving at correct code much faster.

**Notes on Figure 13.34. Point location.** The brief but logically complex function in Figure 13.32 is called from the `testPoints()` function in Figure 13.32.

**The function header.**

- The two parameters are `struct` types, the return value is an enumerated type.
- We use short names for the parameters because the task of this function requires using logical expressions with several clauses. Short names let us write the entire expression on one line in most cases.

**Testing for containment.**

- The logic of this function could be organized in many different ways. We choose to ask a series of questions, where each question completely characterizes one of the locations.
- If a case tests `true`, the program stores the corresponding position code in an `InType` variable named `place`. Otherwise, it continues the testing.

Given a rectangle and a point, return a code for the position of the point relative to the rectangle. This function is called from the `testPoints()` function in Figure 13.32.

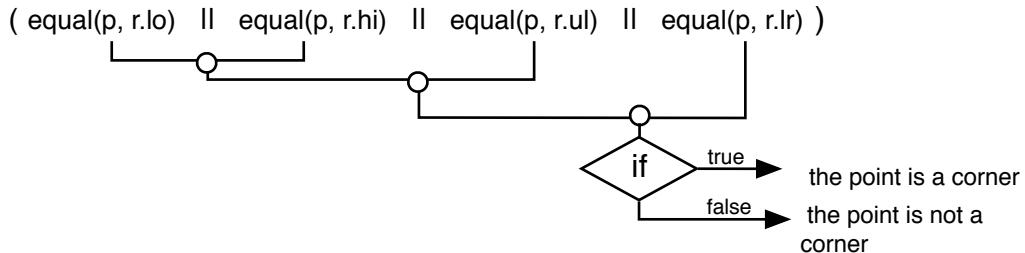
```

InType locate( RectT r, PointT p ) { // Where is p with respect to r?
    InType place; // Set to out, in, corner, or side.
    if (over(p, r.hi) || under(p, r.lo) || left(p, r.lo) || right(p, r.hi))
        place = P_OUT;
    else if (over(p, r.lo) && under(p, r.hi) && right(p, r.lo) && left(p, r.hi))
        place = P_IN;
    else if (equal(p, r.lo) || equal(p, r.hi) || equal(p, r.ul) || equal(p, r.lr))
        place = P_CORNER;
    else place = P_SIDE;
    return place;
}

```

**Figure 13.34.** Point location.

- Each test is a logical expression with four clauses that compares the two coordinates of  $p$  to the four coordinates of  $r$ .
  - The three tests use one logical operator repeatedly. They will be parsed and executed in a strictly left-to-right order using lazy evaluation:



- The most complicated case to identify occurs when a point lies on one of the sides of the rectangle. A logical expression to identify this case would be even longer than the three other expressions. However, by leaving this case until last, we can identify it by the process of elimination. The final `else` clause handles a point on a side.

**Program testing.** The test plan presented in Figure 13.27 tests all possible kinds of rectangles (ordinary and degenerate) with points in all possible relationships to each rectangle. We show some, but not all, the test results here.

With an ordinary rectangle, the output will appear as follows. The first six lines are from `main()` and the point prompt and processing are from `testPoints()`. The final question is again from `main()`.

Given a rectangle with a side parallel to the x axis and a series of points on the xy plane, say where each point lies in relation to the rectangle.

```
Enter x and y for lower left corner: 4 -2  
Enter x and y for upper right corner: 14 4
```

Using lower left = (4, -2), upper right = (14, 4).  
Please enter a series of points when prompted.

Enter x and y (4 -2 to quit): 16 3

The point (16, 3) is outside the rectangle.

```

Enter x and y (4 -2 to quit): 14 -2
The point (14, -2) is at a corner of the rectangle.

Enter x and y (4 -2 to quit): 6 4
The point (6, 4) is on a side of the rectangle.

Enter x and y (4 -2 to quit): 5 -2
The point (5, -2) is on a side of the rectangle.

Enter x and y (4 -2 to quit): 4 2
The point (4, 2) is on a side of the rectangle.

Enter x and y (4 -2 to quit): 14 -1
The point (14, -1) is on a side of the rectangle.

Enter x and y (4 -2 to quit): 8 2
The point (8, 2) is inside the rectangle.

Enter x and y (4 -2 to quit): 4 -2
The point (4, -2) is at a corner of the rectangle.

Do you want to test another rectangle (y/n)? y

```

A specified requirement is that the program perform sensibly with a “rectangle” that is a straight line. We tested such a degenerate rectangle; an excerpt from the output is

```

Enter x and y for lower left corner: 1 0
Enter x and y for upper right corner: 1 3
Using lower left = (1, 0), upper right = (1, 3).
Please enter a series of points when prompted.

Enter x and y (1 0 to quit): 1 3
The point (1, 3) is at a corner of the rectangle.

Enter x and y (1 0 to quit): 2 1
The point (2, 1) is outside the rectangle.

Enter x and y (1 0 to quit): 1 2
The point (1, 2) is on a side of the rectangle.

Enter x and y (1 0 to quit): 1 0
The point (1, 0) is at a corner of the rectangle.

```

A degenerate rectangle that is just a single point also gives sensible output:

```

Enter x and y for lower left corner: 1 3
Enter x and y for upper right corner: 1 3
Using lower left = (1, 3), upper right = (1, 3).
Please enter a series of points when prompted.

Enter x and y (1 3 to quit): 2 0
The point (2, 0) is outside the rectangle.

Enter x and y (1 3 to quit): 1 3
The point (1, 3) is at a corner of the rectangle.

```

## 13.7 Points in a Rectangle written in C++

This C++ implementation is like the C implementation in many ways:

- The basic organization is the same, with three files: the main program and a header file and code file for the two classes.
- Every code file (.c or .cpp) needs to include its matching header file (.h or .hpp). The declarations in the header file are separated from the code because the header file must also be included by any client modules. The header information becomes *common knowledge* that is needed to allow two modules to be linked together and work properly.

```

1 // Figure 13.35: C++ version of points in a rectangle.           main.cpp
2 // -----
3 #include <cctype>
4 #include "tools.hpp"
5 #include "rect.hpp"
6
7 // -----
8 int main( void ) {
9     char again;          // For user response in test_points loop.
10    int xLo, yLo, xHi, yHi; // Input for the coordinates of a rectangle.
11
12    banner();
13    cout <<" Given a rectangle with a side parallel to the x axis \n"
14        " and a series of points on the integer xy plane,\n"
15        " say where each point lies in relation to the rectangle.\n";
16
17    do {
18        cout <<"\n Enter x and y for lower left corner: ";
19        cin >>xLo >>yLo ;
20        cout <<" Enter x and y for upper right corner: ";
21        cin >>xHi >>yHi ;
22
23        RectT rect( xLo, yLo, xHi, yHi );           // Create the rectangle.
24        rect.testPoints();                         // Create and test a series of points.
25
26        cout <<"\n Do you want to test another rectangle (y/n)? ";
27        cin >> again;
28    } while (tolower( again ) != 'n');
29    return 0;
30 }
```

---

**Figure 13.35. Main function in C++ for points and rectangles.**

- The program logic is the same, including the treatment of mixed up and degenerate input cases.
- The C++ and C types create identical data objects.
- The output from the program is the same.

There are also several major ways that this C++ implementation differs from the C implementation. Discussion will focus on the differences.

- Data members are private.
- C++ classes have function members as well as data members. A function is called using the name of a class object.
- Inline functions are introduced. An inline function is completely defined in the header file. When this function is called, the compiler will replace the call by the body of the function. It will not do a jump-to-subroutine and return. This improves performance at run time. This is only recommended for short functions.
- The C++ classes have constructors and print functions.
- Throughout the code, **const** is used in the modern manner to limit opportunity for undetected program bugs.
- The I/O system is different.
- The syntax for **enum** declarations is simpler than C, but the usage is the same.

**Notes on Figure 13.35: The main function in C++.** Little comment is needed for this function because, except for the way I/O is done, it is nearly identical to the C version. It prints a title and has a query loop to allow entry of a series of rectangles. For each rectangle, the four number read as input are used to construct a rectangle, and the rectangle is used to call the `testPoints()` function.

**Notes on Figure 13.36: C++ classes for points and rectangles.** Compare this to the C header file in Figure 13.29.

**Line 33:** All of the C++ programs in this text will use functions from the `tools` module. This header file also includes all the other standard C++ library headers that we are likely to need. The functions that are prototyped here are defined in the file `rect.cpp` in Figure 13.38

**Line 34:** This enumeration is simpler than the C version. We don't need `typedef` and a `typedef` name at the end. The name we will use is the one that follows the keyword `enum`.

**Lines 37 through 49:** This is the class declaration for the point type, `PointT`. In addition to data members, it lists prototypes or definitions for all the functions that relate to points.

- Line 41 is a complete definition of a default constructor that will initialize all members of a new object to 0's. A default constructor is needed to create an uninitialized class object.
- Line 42 is the constructor with parameters that will be used to initialize points when the data is input.
- Lines 44 ... 48 are complete definitions of inline functions. The .cpp file does not contain anything for these functions. Any function that fits on one line should be defined this way.

```

31 // Figure 13.36: Point and Rectangle classes in C++           rect.hpp
32 // -----
33 #include "tools.hpp"
34 enum InType{ P_IN, P_SIDE, P_CORNER, P_OUT };
35
36 // -----
37 class PointT {
38 private:
39     int x, y;                                // x and y coordinates of a point.
40 public:
41     PointT() = default;                      // Default constructor.
42     PointT( int x, int y );                  // Constructor with parameters.
43     void print() const ;
44     bool equal ( PointT p2 ) const { return x == p2.x && y == p2.y; }
45     bool over ( PointT p2 ) const { return x > p2.x; }
46     bool under ( PointT p2 ) const { return x < p2.x; }
47     bool left ( PointT p2 ) const { return y < p2.y; }
48     bool right ( PointT p2 ) const { return y > p2.y; }
49 };
50
51 // -----
52 class RectT {
53 private:
54     PointT lo;        // bottom left corner of rectangle in the x-y plane
55     PointT hi;        // top right corner of rectangle in the x-y plane
56     PointT ul;        // top left corner of rectangle in the x-y plane
57     PointT lr;        // bottom right corner of rectangle in the x-y plane
58 public:
59     RectT( int xLo, int yLo, int xHi, int yHi );
60     void testPoints();
61     InType locate( PointT P ) const ;
62     void print() const;
63 };

```

Figure 13.36. The C++ class declarations for points and rectangles.

```

64 // Figure 13.37: Functions for the Point class.          point.cpp
65 // -----
66 #include "rect.hpp"
67 const char* answer[4] = {"inside", "on a side of", "at a corner of", "outside"};
68 // -----
69 PointT::PointT( int x, int y ) {
70     this->x = x;
71     this->y = y;
72 }
73 // -----
74 void PointT::print() const {
75     cout << "(" <<x <<, " <<y <<") ";
76 }
```

---

**Figure 13.37.** C++ functions for points.

- The modifier `const` is used on after the closing `)` and before the opening `{` for six of these functions. A `const` in this position means that the function will not change the implied parameter. Further, if it calls other functions, *they* will not be permitted to change the implied parameter, either.
- Note that the careful layout improves readability.

**Lines 22 through 33:** This is the class declaration for the rectangle type, `RectT`.

- Lines 24–27 declare data members to represent the four corners of the rectangle. A comment is given on each to make its purpose clear.
- Line 29 is the constructor with parameters that will be used to initialize rectangles when the data is input.
- Lines 31 and 32 are `const` functions. The `const` keyword after the parameter list prevents any change to the implied parameter.

#### Notes on Figure 13.37: C++ functions for points.

**Line 66:** The code file `point.cpp` includes its the header file that contains the class declaration: `rect.hpp`, which is also included by the `main.cpp`. The header information becomes *common knowledge* that is needed to allow these two modules to work together.

**Line 67:** As in the C version, we use an array of constant strings to produce readable, English-language answers. This is an array of `const char*` strings, not of C++ strings, because there is no need for the “growing power” of the C++ string.

**Lines 69...72:** comprise the `PointT` constructor with parameters. This kind of constructor uses the parameters to initialize the newly allocate object. In this case, there is one parameter per data member and it is very natural to give them the same names. To break the ambiguity, we refer to the data member named `x` as `this->x` and to the parameter as `x`. This is a common way to write a constructor.

**Lines 74...76:** The `print` function in this class has no analog in the C version. However, a class is supposed to be the expert on its members, and that includes knowing how to print them. So we define a `print()` function that formats the data members nicely. Note that it does not put a newline on the end of the line. It is better to let the caller decide whether that newline is wanted.

#### Notes on Figure 13.38: C++ functions for rectangles.

**Line 79: The include.** As in the C version, every `.cpp` file must include its matching header file. This should be the only `#include` command in the file. All other includes should be in the header file.

---

```

77 // Figure 13.38: Functions for the Rectangle class.           rect.cpp
78 // -----
79 #include "rect.hpp"
80
81 // Read, validate, and store the coordinates of a rectangle.
82 RectT::RectT(int xLo, int yLo, int xHi, int yHi) {
83     // If corners were not entered correctly, swap coordinates.
84     if (xLo > xHi) swap( xLo, xHi );
85     if (yLo > yHi) swap( yLo, yHi );
86
87     lo = PointT( xLo, yLo );          // These 2 corners define the rectangle.
88     hi = PointT( xHi, yHi );
89     ul = PointT( xHi, yLo );         // Used to test if point is on a corner.
90     lr = PointT( xLo, yHi );
91 }
92
93 // -----
94 // Analyze the position of a point with respect to a rectangle in the xy plane.
95 //
96 void RectT::testPoints() {
97     InType where;      // Position of point relative to rectangle.
98     PointT pt;
99     int x, y;          // For inputting coordinates.
100
101    cout <<" Please enter a series of points when prompted." ;
102    do {
103        cout <<"\n Enter x and y ";
104        lo.print();
105        cout <<" to quit.  ";
106        cin >>x >>y ;
107        pt = PointT(x, y);
108        where = locate( pt );
109        pt.print();
110        cout <<" is " <<answer[where] <<" the rectangle.\n";
111    } while ( ! pt.equal( lo ) );
112 }
113
114 // -----
115 // Given a point and a rectangle, return a code for the position of the point
116 // relative to the rectangle.
117 InType RectT::
118 locate( PointT p ) const {
119     InType place;                  // Set to out, in, corner, or side.
120
121     if (p.over(hi) || p.under(lo) || p.left(lo) || p.right(hi)) place = P_OUT;
122     else if (p.over(lo) && p.under(hi) && p.right(lo) && p.left(hi)) place = P_IN;
123     else if (p.equal(lo)|| p.equal(hi) || p.equal(ul) || p.equal(lr))place = P_CORNER;
124     else place = P_SIDE;
125     return place;
126 }
127
128 // -----
129 // Print the coordinates of a rectangle.
130 void RectT::
131 print() const {
132     cout <<"\n Lower left corner = ";
133     lo.print();
134     cout <<" Upper right corner = ";
135     hi.print();
136 }
```

---

Figure 13.38. C++ functions for rectangles.

***Lines 82...91: The constructor.***

- Lines 84...85 check for and correct data that was entered wrong. The C++ `<algorithm>` library has an implementation of swap. There is no need to define it here in the rectangle program.
- Lines 87...90 use the input values to construct points, and assign those points to the data members of the class.
- Line 104 calls a function in the point class. The implied parameter will be the point named `lo`.
- Line 108 calls a function in the current class, `rectangle`. We do not need to write an object name in front of the function name when a function is called from another function in its own class. The implied parameter in `locate()` will be the same object that is the implied parameter in `testPoints()`.
- This is different from the C version. In C, there is no privacy, and the `makeRect()` function simply reached into the points and set the `x` and the `y`. In C++, those parts are private and the rectangle constructor cannot access them. So it uses the input data to call the point constructor and initialize the points. This is OO: privacy matters.

***Lines 96...112: testPoints().***

- This function operates exactly like the C version. It reads a point, locates it w.r.t the rectangle, and prints the answer. Data entry and analysis is repeated until the sentinel values (the coordinates of the lower left corner) are entered.
- Lines 106 and 107 input `x` and `y` coordinates then call the point constructor to initialize a new point, which is stored in the variable `pt`. We go through the point constructor because the coordinates are private within the point and we cannot do this job the way it was done in C, by reading directly into `pt.x` and `pt.y`.
- Lines 109 and 110 print the output. In C, this can be done in one line. C++ also permits it to be done in one line by using an operator definition. (This will be covered later in the text.)

***Lines 117...126: locate().***

- Note that the return type and class name are on line 117 and the function name on line 118. This puts the function name on the left margin where is it easy to find. This style makes a lot of sense in C++, increasingly so as the programmer starts using more complex types.
- This function operates exactly like the C version. It reads a point, locates it w.r.t the rectangle, and prints the answer. Data entry and analysis is repeated until the sentinel values (the coordinates of the lower left corner) are entered.
- Lines 106 and 107 input `x` and `y` coordinates then call the point constructor to initialize a new point, which is stored in the variable `pt`. We go through the point constructor because the coordinates are private within the point and we cannot do this job the way it was done in C, by reading directly into `pt.x` and `pt.y`.
- Lines 109 and 110 print the output. In C, this can be done in one line. C++ also permits it to be done in one line by using an operator definition. This is too advanced now, but will be covered later in the text.

***Lines 117...126: print().***

- Every C++ class should have a print function. It is essential for debugging.
- This code for printing the rectangle would be a 1-line function if we provided definitions for `<<` on rectangles and points.

## 13.8 What You Should Remember

### 13.8.1 Major Concepts

- An enumerated type specification allows you to define a set of related symbolic constants.
- Use `typedef` declarations to name enumerations and struct types in C. They are not necessary in C++ for these purposes.
- Types `char` and `bool` are the most commonly used enumerated types.
- A `struct` or a `class` can be used to represent an object with several properties. Each data member of the structure represents one property and is given a name that reflects its meaning.

- A `struct` or `class` type specification does not create an object; it creates a pattern from which future objects may be created. Such patterns cannot contain initializations in C, but they can in C++.
- The basic operations on objects include member access, assignment, use as function parameters and return values, and hierarchical object construction.
- Structured types can be combined with each other and with arrays to represent arbitrarily complex hierarchical objects. A table of data can be represented as an array of structures.
- An entire structured object can be used in the same ways as a simple object, with three exceptions. Input, output, and comparison of structured objects must be accomplished by operating on the individual components.
- Design principles.
  1. A class protects its members. It keeps them private and ensures that the data stored in an object is internally consistent and meaningful at all times.
  2. A class should be the expert on its own members. Implement all the functions needed to use the class.
  3. Delegate the activity to the expert. Don't try to do things in the wrong class.
  4. Creation and deletion. With dynamic allocation, new objects should be created by the class that will contain them, and that class is responsible for deleting them when they are no longer needed.
  5. The functions that belong in a class are those that deal with one class instance.
  6. Keep it short and simple. If member names are long, code is more difficult to write, read, and debug.

### 13.8.2 Programming Style

**Enumerations.** Use enumerations wisely. Use an enumeration any time you have a modest sized set of codes to define. It might be a set of error codes, conditions, or codes relating to the data. Use `#define` for isolated constants. Often, it is wise to include one item in the enumeration to use when an unexpected error occurs. Names of the enumeration constants should reflect their meaning, as with any constant or variable. For example, an enumeration for “gender” might be `enum { male, female, unknown }`.

**Truth values.** Many functions and operators return truth values (`false` or `true`). Use the proper symbols for truth values; avoid using 0 and 1. Learn to test truth values directly rather than making an unnecessary comparison; for example, write `if (errorFlag)` rather than `if (errorFlag == true)`.

**Structure declarations.** Declare all C structures within `typedef` declarations. This simplifies their use and makes a program less error prone. Using a tag name in addition, in a `struct` specification, is an advantage with some on-line debuggers.

**Long-winded names.** The names of the members of a `struct` or `class` should be as brief as possible while still being clear. They must make sense in the context of a multipart name, but they need not make sense in isolation.

**Privacy.** In a `class` keep your data members private. Minimize the number of accessors (“getters”) you define and provide a mutator (“setter”) only in unusual cases.

**Using `typedef` wisely.** Type declarations can be overused or underused; try to avoid both pitfalls. Use of `typedef` should generally follow these guidelines:

- Use `typedef` with a `struct` type for a collection of related data items that will be used together and passed to functions as a group.
- Any type named using `typedef` should correspond to some concept with an independent meaning in the real world; for example, a point or a rectangle.
- Each `typedef` should make your code easier to understand, not more convoluted. Names of types should be concise, clear, and not too similar to names of objects.

**Call by value and return by value.** Some structured objects are quite large. In such cases, using call by value for a function parameter consumes a substantial amount of run time during the function call because the entire structure must be copied into the parameter. This becomes significant because functions generally are used for input, output, and comparison of structures and so are called frequently. To increase program efficiency for functions that process large structures, avoid both call by value and return by value. Replace these techniques by call by address, with a `const` modifier, where appropriate.

In this chapter, both large and small structures have been declared and used. The point structure is small and it is appropriate to use call-by-value with it. The lumber structure is much larger and should be passed by address.

**Array vs. structure.** Use a structure to represent an aggregate type with heterogeneous parts. For example, the `LumberT` in Figure 13.9 is defined as a structure because its five members serve different purposes and have different types. If all parts of an aggregate are of the same type, we can choose between using an array and using a structure to represent it. In this case, the programmer should ask, “Do I think of the parts as being all alike or do the various parts have different purposes and must they be processed uniquely?”

For example, assume you want to measure the temperature once each hour and keep a list of temperatures for the day. This is best modeled by an array, because the temperatures have a uniform meaning and will be processed more-or-less in the same manner, likely as a group.

Sometimes either technique can be used. For example, the rectangle defined in Figure 13.30 could have been defined as an array of eight integers. However, each `int` in the `RectT` structure has a different meaning, and the numbers in the first pair must be less than the numbers in the second pair. Giving unique symbolic names to the lower and upper corners, as well as to the *x* and *y* coordinates of each corner, makes the program easier to write and understand.

**Parallel arrays vs. an array of structures.** These two ways to organize a list of data objects are syntactically quite different but similar in purpose. Most applications based on one could use the other instead. An array of structures provides a better model for the data in a table. Parallel arrays are well suited for menu processing and masking; they should be used when the content and purpose the items in one array is only loosely related to the content and purpose of the other, or when one of the arrays must be used independently of the other.

### 13.8.3 Sticky Points and Common Errors

**Enumeration constants are not strings.** The words that you write in an `enum` definition are literal symbols, not strings. Use them directly; do not enclose them in quotes. Because of this, you cannot input or output the symbols directly; you must use the underlying integer values for these purposes.

**Call by value.** If a parameter is not a pointer (and not an array), changes made to that parameter are not passed back to the caller. Be sure to use call by address if any part of a structured parameter is modified by the function.

**The dot and arrow operators.** Two C operators are used to access members of structures: the dot and the arrow. If `obj` is the name of a structured variable, then we access its parts using the dot. If `p` is a pointer to a structured variable, we use the arrow.

Even so, programmers sometimes become confused because the same object is accessed sometimes with one operator and sometimes with the other. For example, suppose a program passes a structured object to a function using call by address. Within the main program, it is a structure and its members are accessed using dot notation. But the function parameter is a pointer to a structure, so its members are accessed within the function using an arrow. Another function might use a call-by-value parameter and would access the object using a dot.

For these reasons, it is necessary to think carefully about each function parameter and decide whether to use call by value or address. This determines the operator to use (dot or arrow) in the function’s body. If you use the wrong one, the compiler will issue an error comment about a type mismatch between the object name and the operator.

**Comparison and copying with pointers.** Remember that a pointer to a structure is a two-part object like a string. If one pointer is assigned or compared to another, only the pointer part is affected, not the underlying structure. To do a structure assignment through a pointer, you must dereference the pointer. To do a structure comparison, you must write a comparison function and call it.

### 13.8.4 New and Revisited Vocabulary

These are the most important terms and concepts presented in this chapter:

enumerated type	member	pointer to a structure
enumeration constant	member name	operations on structures
type declaration	member declaration	comparing structures
type <b>bool</b>	structured initializer	hierarchical structures
<b>true</b> and <b>false</b>	tag name	array of structures
error codes	<b>typedef</b> name	parallel arrays
aggregate type	selection expression	delegating work
structure	dot operator (member access)	expert principle
class	arrow operator (member access)	privacy principle
object	call by value or address	sentinel value

The following keywords, C library functions, and constants were discussed or used in this chapter:

<b>struct</b>	<b>typedef</b>	* (dereference)
<b>class</b>	<b>private</b>	-> (dereference and member access)
<b>enum</b>	<b>public</b>	. (member access)

### 13.8.5 Where to Find More Information

- The semantics and usage of type **bool** in C++ are covered in the *C++: The Complete Reference* by *Herbert Schildt*. Refer to the index for page references in the current edition.
- The semantics and usage of type **boolean** in Java are covered in Chapter 4.2.5. of the *Java Language Specification* which can be downloaded from

[http://java.sun.com/docs/books/jls/second\\_edition/html/j.title.doc.html](http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html)

- Self-referential structure definitions create types in which one structure member is a pointer to an object of the **struct** type currently being defined. These types are presented in Chapter ??, with extensive examples of their use in building linked lists.
- Chapter ??, Figures ??, ??, and ?? illustrate recursive type definitions in which the tag name is required.
- The **sizeof** operator is also discussed in Figures 4.11, 10.5, and 12.16. It is used for memory management in Chapter 16.



# Chapter 14

## Streams and Files

We have been using the input functions (`scanf()`, `cin >>`) and output functions (`puts()`, `printf()` and `cout <<`) without explaining much about the streams they read from and write to. In this chapter, we introduce the concept of streams: the predefined streams and programmer-defined streams. We also describe what a stream is and how to interact successfully with streams. We present stream management techniques, discuss file input and output, consider I/O errors, and present examples of a crash-resistant code.

Even with no compile-time errors and all the calculations correct, a program's output can be wrong because of run-time errors during the input or output stages. Such errors can be caused by format errors, incorrect data entry, or human carelessness. Hardware devices also occasionally fail. Three ways to detect certain input errors have been used since the beginning of this text:

1. Echo the input data to see what the program actually is processing for its computations. The values displayed may not be the values the user intended to enter.
2. Use `if` or `while` statements to test whether the input values are within acceptable ranges.
3. Write a test plan that lists important categories of input with the corresponding expected output. Use these examples of input to test the program and inspect the output to ensure correctness.

We describe one more important method of detecting input errors and show a variety of ways to recover from these mistakes.

The conceptual material is the same for C and C++, but of course, the function names and syntax are different. In this chapter, we present the C++ I/O system. The same material, explained and written in C is in Appendix ??.

### 14.1 Streams and Buffers

To understand the complexities of I/O, it is necessary to know about streams and buffers. In this section we describe these basic components of the I/O system.

#### 14.1.1 Stream I/O

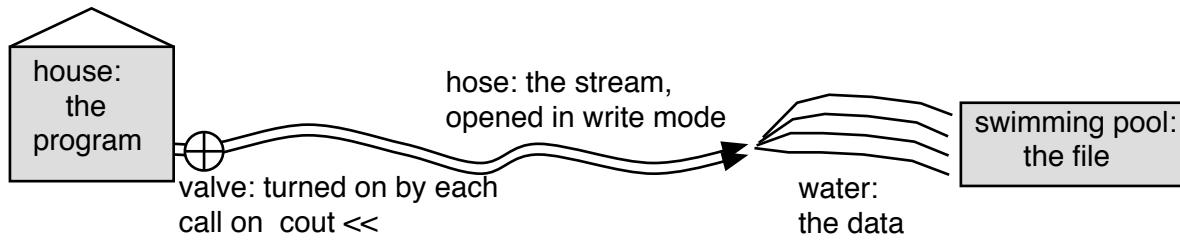
The input or output of a program is a sequence of data bytes that comes from or goes to a file, the keyboard, the video screen, a network socket,<sup>1</sup> or, possibly, other devices. A stream is the conduit that connects a program to a device and carries the data into or out of the program.

You can think of a stream as being like a garden hose. To use a hose, we must find an available one or purchase a new one. Before water can flow, the hose must be attached to a source of water on one end and the other end must be aimed at a destination for the water. Then, when the valve is opened, water flows from source to destination (see Figure 14.1). When we open a stream for reading, it carries information from the source to our program. A stream opened in write or append mode carries data from the program to a

---

<sup>1</sup>A network socket is used to connect an input or output stream to a network.

An output stream connects a program to a file or an output device. Calling an output function is like opening the valve: It lets the data flow through the stream to their destination.



**Figure 14.1.** A stream carries data.

destination. Calling an input function to read the data or an output function to write them, is like turning on the valve and letting the water flow.

**Why we need streams.** Streams were invented to serve several important ends:

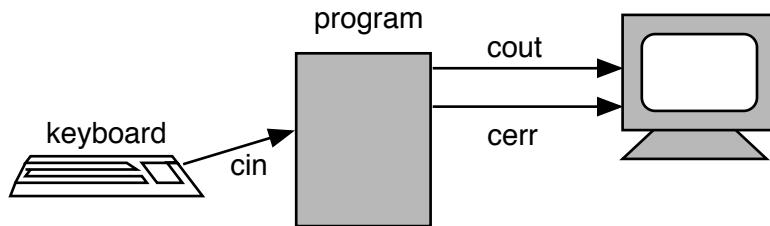
- A stream provides a standard interface for using diverse kinds of equipment. Disk files, sockets, keyboards, and video screens have very different properties, different needs, and different machine language. All of these differences are hidden by the stream interface, allowing the programmer to use one set of functions to read or write from any of these devices.
- Streams provide buffering. An input stream buffer stores blocks of input characters and delivers them to the program one line or one character at a time. An output stream buffer stores the characters that come from `puts()`, `printf()` and `cout <<` until enough have been collected to write a whole line on the screen or a whole block on the disk.
- Error flags. A stream is a data structure used to keeps track of the current state of the I/O system. It knows the location of the source or target device or file. It knows how much has been read from or written to the device into the buffer. It knows how much data has been read from or stored into the buffer. Finally, it contains status flags that can be tested to detect errors.

**Standard streams in C++.** Four streams are predefined in C++, as listed in Figure 14.2. The standard streams are opened automatically when you program begins execution and are closed automatically when it terminates. `cin` is connected, by default, to the keyboard; `cout` and `cerr` are connected by default to the monitor screen, as shown in Figure 14.3. These default connections provide excellent support for interactive input and output.

- The standard C++ input stream is `cin`. Normally, `cin` is connected to the user's keyboard, although most systems permit it to be redirected so that the data come from a file instead.
- The standard C++ output stream is `cout`. Normally, `cin` is connected to the user's video screen, although most systems permit it to be redirected so that data go to a file instead.
- The standard error stream, `cerr`, is used by the runtime system to display error comments, although any program can write output, instructions, and error messages into `cerr`. The error stream normally is connected to the user's video screen. Although it can be redirected, it seldom is.

Purpose	Stream name C++
Input from the keyboard	<code>cin</code>
Output to the screen	<code>cout</code>
Error output on screen	<code>cerr</code>
Logging to a file	<code>clog</code>

**Figure 14.2.** Standard streams.



**Figure 14.3.** The three default streams.

### 14.1.2 Buffering

Water moves through a garden hose in a continuous stream. In contrast, the C++ standard input and output streams are buffered, causing the data to flow through the stream in blocks. A **buffer** is a sequence of memory locations that is part of the stream. It lies between the producer and the consumer of the data, temporarily storing data after they are produced and before they are consumed, so that the characteristics of the I/O devices can be matched to the operation of the program.

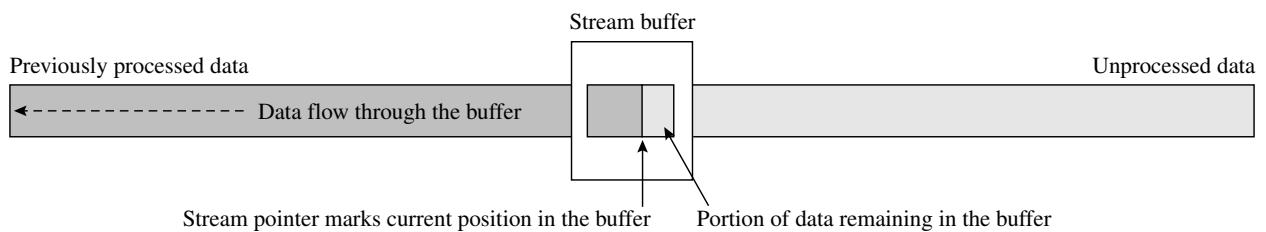
The system reads or writes the data in blocks, through the stream, even if the user inputs or outputs data one item at a time. This buffering normally is transparent to the programmer. However, there are some curious and, perhaps, tricky aspects of C++ input and output that cannot be understood without knowing about buffers.

**Input buffering.** An input stream is illustrated in Figure 14.4. The data in a stream flow through the buffer in blocks. At any time, part of the data in the buffer already will have been read by the program and part remains waiting to be read in the future.

An input stream delivers an ordered sequence of bytes from its source, to the program, on demand. When the buffer for an input stream is empty and the user calls for more input, an entire unit of data will be transferred from the source into the buffer. The size of this unit will vary according to the data source. For example, if the stream is connected to a disk file, at least one disk cluster (often 1,024 or 2,048 characters) will be moved at a time. When that data block is used up, another will be retrieved.

Keyboard input actually goes through two buffers. The operating system provides one level of buffering. Keystrokes go into a system keyboard buffer and remain there until you press Enter. If you make an error, you can use the Backspace key to correct it. Using the Enter key causes the line to go from the keyboard buffer into the stream buffer. Then your program takes the data from the stream buffer one item at a time.

**Output buffering and flushing.** An output stream delivers an ordered sequence of bytes from the program to a device. Items will go into a buffer whenever an output function is called. The data then are transferred to the device as a block, whenever the output stream is flushed, which may occur for several reasons. Information written to `cout` normally stays in the buffer until a newline character is written or the program switches from producing output to requesting input. At that time, the contents of the buffer are flushed to the screen and



**Figure 14.4.** An input buffer is a window on the data.

become visible to the user. For other output streams, the system will send the output to its destination when the buffer is full. All output buffers are flushed automatically when the program exits.

A program also can give an explicit command to flush the buffer, of an output stream<sup>2</sup>, which sends the data in that stream's buffer to its destination immediately. Flush is defined in the class `ostream` as a manipulator: `streamName << flush;`

In contrast to `cout`, `cerr` is an **unbuffered** output **stream**. Messages sent to it are passed on to their destination immediately. When both standard streams are directed to the same video screen, it is possible for the output to appear in a different order than that which was written. This could happen when a line sent to `cout` does not end in a newline character and is held in the buffer indefinitely, until a newline is written or the program ends. Other material sent to `cerr` during this holding period will appear on the screen first.

### 14.1.3 Formatted and Unformatted I/O.

**Input.** When data is read from the keyboard or from a text file, it comes into the stream as a series of characters. If a single character, an array of chars or a string is the end goal, the raw data is just stored in the variable. This is unformatted input.

However, if the program is using `>>` and the code to read input into a numeric variable, an additional step, number conversion, must happen before storing the result in the program's variable. First, the array of characters stored in the stream buffer must be parsed. This involves skipping leading whitespace to find the beginning of the ASCII-number. Then the conversion happens, as follows:

1. Start with `ch`, the first numeric character. Set `result=0`;
2. Subtract '0' (the character representing zero) from `ch`. This gives a binary number between 0 and 9.
3. Add the answer to `result`.
4. Increment your loop counter and read the next character, `ch`.
5. If `ch` is not a digit, leave the loop. If `ch` is a digit, set `result *= 10`.
6. Repeat from step 2.
7. When this loop ends, `result` is the binary integer that corresponds to the original text string.

If the input variable is a `float` or a `double`, two additional actions happen during conversion:

1. The position of the decimal point, if any, is noted, and the number of digits after the decimal point,  $n$ , is counted.
2. When this loop ends, `result` is divided by  $10^n$ .

The way whitespace in the input is handled can be controlled by using stream manipulators:

**Formatted Output.** Number conversion also happens during output when the program calls `>>` to output a numeric variable. Stream manipulators are used to control the formatting, including:

<code>flush</code>	Flush the stream buffer.
<code>endl</code>	End the line and flush it for interactive streams.
<code>hex and dec</code>	Put the stream in hexadecimal or decimal mode.
<code>setw( n )</code>	Output the value in a field that is $n$ columns wide
<code>right and left</code>	Display the value at right or left edge of the field.
<code>setfill( ch )</code>	Use a fill character other than spaces.
<code>setprecision( n )</code>	Show $n$ places after the decimal point.
<code>fixed</code>	Display all float and double values in fixed point notation.
<code>scientific</code>	Display float and double values in scientific notation.

Other manipulators are define in the standard classes `ios`, `iomanip`, and `ostream`. Examples of the use of several of these manipulators will be given in the programs in this chapter.

<sup>2</sup>Flushing is not predefined for input streams but a definition is given in the tools library.

---

We show how to define streams, open them, and close them.

```
#include <iostream>      // For the standard streams.
#include <fstream>       // For file streams.
#include <iomanip>        // A set of stream management and formatting tools.
#include <sstream>        // For string-streams.

ifstream fIn( "prog4.in" );
ifstream sIn;
sIn.open( "a:\\cs110\\my.in" );

ofstream sOut( "myfolder/my.out" );
ofstream fOut( "prog4.out", ios::append );
ofstream bOut( "image1.out", ios::binary );

fIn.close();
fOut.close();
```

---

Figure 14.5. Opening streams in C++.

## 14.2 Programmer-Defined Streams

A programmer who wishes to use a file must open a stream for that file. Opening a stream creates the stream buffer and connects the stream to a file.

Streams can be opened for input, output or both and they can be created in text mode or binary mode. **Text files** (in character codes such as ASCII or Unicode) are used for most purposes, including program source files and many data files. **Binary files** are used for executable code, bit images, and the compressed, archived versions of text files. In this chapter, we concentrate on using text files; binary data files are explored in Chapter 18.

### 14.2.1 Defining and Using Streams

**Types of streams.** Streams can be for input (`istream`), output (`ostream`), or both (`iostream`). The type used to declare the stream name determines what kind of stream it is. In this chapter we will use `istreams` and `ostreams`<sup>3</sup>.

Figure 14.5 illustrate how streams can be declared and opened in C++, resulting in the configuration shown in Figure 14.6. Opening a stream creates a data structure that is managed by the system, not by the programmer. This structure includes memory for the stream's buffer, a pointer to the current position in the buffer, flags for error and end-of-file conditions (discussed later), and anything else needed to maintain and process the stream. The programmer never accesses the stream object directly, so it is not necessary to know exactly how it is implemented.

**Notes on Figure 14.5: Opening streams.** We declare five streams in this Figure: `fIn`, `fOut`, `sIn`, `sOut`, and `bOut`. When opened properly, these streams will allow us access to five different files. Figure 14.6 shows the configuration that would be created if only `sIn` and `sOut` were opened.

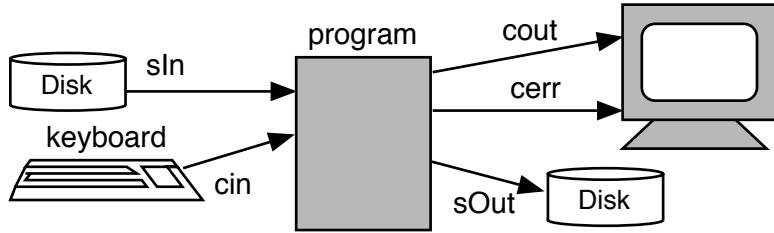
**First Box: Include files.** One to four header files must be included to use streams in C++.

- The `<iostream>` header gives access to the four predefined streams and to the classes used to define them: `<istream>`, `<ostream>` and `<ios>`.
- The `<fstream>` header lets us define file-streams.
- The `<iomanip>` header gives access to a set of file management tools involved with formatting, octal and hexadecimal I/O, whitespace, and flushing buffers.

---

<sup>3</sup>The use of `iostreams` is beyond the scope of this text.

This shows the three default stream assignments, plus the streams `sIn` and `sOut`, which were declared and opened in Figure 14.5.



**Figure 14.6. Programmer-defined streams.**

- The `<sstream>` header allows us to use sophisticated input parsing and validation techniques. It is useful for applications that must process complex data. Briefly, a string is used as a temporary place to store a line of input while it is analyzed (`istringstream`) or created (`ostringstream`), and normal stream syntax is used to get data into or out of the string. Examples will be given at the end of this chapter.

**Second Box: Declaring and opening input streams.** In C++, a stream can be declared and opened in the same line or separately, on two lines.

- The first line of code both declares an input stream and opens it. This is the normal way to open a file when the name of the file is known to the programmer.
- Alternatively, a stream can be declared with no initializer and opened in a later line, as in the second and third lines of code. This method is used when the name of the stream is not known prior to runtime.
- This box opens two input streams. For one, the parameter is just a file name. At run time, the file needs to be in the same directory as the executable program. This method of opening a file is strongly preferred because it is portable from one machine to another.

For the other stream, a full pathname is given for the file. This works, but is not portable. Normally, no two file directories have the same paths in them.

**Third box: Output streams.** The one-line declaration with initialization, and the two-line declaration and separate call on `open()` can be used to open all kinds of streams. In this box, we show output streams with some additional opening options.

- `ofstreams` are opened in output mode. If the named file does not already exist in the named directory, it will be created. If it does exist, the old version will be destroyed. For this reason, some caution is advised; when selecting a name for an output file, it should not duplicate the name of another file in the same directory.
- On the first line, we open `sOut` and attach it to the file `my.out` in the folder named `myfolder`.
- The second line opens a stream `fOut` in and attach it to the file `prog4.out`. Material sent to `fOut` will be appended to the material that is already in `prog4.out`. If the file does not yet exist, it is created and the stream shifts to write mode. Use append mode to capture the output of several runs of your program.
- The third stream in this box, `bOut`, is opened for output in binary mode and attached to a file that will receive a binary image.
- `ios::` is the name of a base class of `iostream`. This class defines things that are common to all streams; it defines constants such as `in`, `out`, `append` and `binary` that control the mode of a stream.

**Fourth box: Closing streams.** All streams are automatically closed when your program terminates, and output buffers are flushed before the stream is closed. Nevertheless, it is good practice in any program to close a stream when you are done with it. Closing a stream releases the buffer's memory space and allows the file to be used by others. This becomes more and more important as the complexity and running time of a program increases.

### 14.2.2 File Management Errors

**Stream-opening errors.** When you open a stream, either in the declaration or by calling `open()`, the result is normally an initialized stream object. If the attempt to open a stream fails, the result is a null pointer.

Several things must be done to open a stream; if any one of them fails, the “opening” action will fail. These include

- An appropriate-sized buffer must be allocated for the stream. The operation fails if space for this buffer is not available in the computer’s memory. If allocation is successful, the address of the buffer is stored as a member of the stream object.
- For input, the operating system must locate the file. If the file is protected, or the name is misspelled, or the path is wrong, or no file of that name exists in the designated directory, `open()` will fail.
- For output, the operating system must check whether a file of that name already exists. If so, it is deleted. Then space must be found on the disk for a new file. This process can fail if the disk is full, the file is protected, or the user lacks permission to write in the specified directory.

Because opening failure is common, it is essential to check for this condition before trying to use the newly opened stream. Otherwise, you are likely to “hang” or crash your program. Such a check is simple enough: call `is_open()` and take appropriate action if a failure is detected. This technique is shown in the next two programs. File-opening errors are handled by calling the function `fatal()`, defined in the tools library.

**Stream-closing errors.** Closing a stream causes the system to flush the stream buffer, do a little bookkeeping, and free the buffer’s memory. It is rare that something will go wrong with this process; a full disk is the main cause for problems. Both `flush()` and `close()` return integer error indicators. However, since trouble occurs infrequently, and at the end of execution, programs rarely check for them. We will not do so in this text.

## 14.3 Stream Output

Once a stream has been properly opened in write or append mode, data can be sent from the program to the stream destination. The syntax for writing to any `ostream` is the same as the syntax for using `cout`. This is illustrated in the next program.

### Notes on Figure 14.8: Writing a file of voltages.

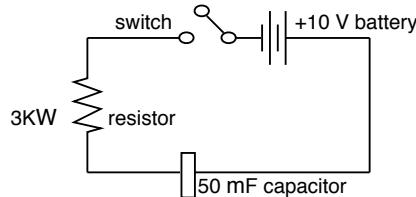
**First box: The name of the input file.** We don’t absolutely need to `#define` the name of the input file. We could simply write it in the code in the fourth box. However, it is good design to put all file names where they can be found and changed easily. Also, this makes it easier to write good error comments, as in box 3. A `#define` command near the top of the program does this for us.

**Second box: Other `#define` commands.** These symbols `step` and `epsilon` are defined by expressions rather than by simple constants. An expression is permitted so long as it involves only known values, not variables. However, because of the way the preprocessor translates `#define`, an expression sometimes can lead to unintended precedence problems. To avoid this, enclose the definition in parentheses, as shown here, or use a `const` variable.

### Third box: declaring and opening the output stream.

- We need to declare a stream for the output; here, we create one named `voltFile`. The name of the stream is arbitrary and need not resemble the actual name of the data file we write.
- We declare the stream `voltFile`, open the file `voltage.dat` in the current default directory. The file will be created if it does not already exist. If it does exist, the former contents will be lost.
- The value `nullptr` is returned by `open()` if any of the steps in creating a stream goes wrong. We check the return value to be sure the open command was successful and abort execution if it was not. We use the same `#define` name in both the open command and the following error report.

**Problem scope:** Calculate the measured voltage across the capacitor in the following circuit as time increases from  $t = 0$  to  $t = 1$  second, in increments of 1/15th second, assuming the switch is closed at time 0. Write the resulting data to a file.



**Input:** None.

**Output required:** The results are stored in the file `voltage.dat`. One data should be written per line for later use by a graphing program.. Each one should have two values,  $t$  and  $v$ , Four decimal places of precision are appropriate.

**Constants:** In this circuit,  $V = 10$  volts,  $R = 3,000$  ohms, and  $C = 50 \times 10^{-6}$  farads. The circuit has a time constant  $\tau$  (tau), which depends on the resistance,  $R$ , and the capacitance,  $C$ , as  $\tau = R \times C = 0.15$  second.

**Formula:** If the switch is closed at time 0, the voltage across the capacitor is given by the following exponentially increasing equation:

$$v(t) = V \times \left(1 - e^{-\frac{t}{\tau}}\right)$$

**Figure 14.7. Specifications: Creating a data table.**

- The function `fatal()` in the tools library provides a succinct and convenient way to print an error message, hold the message on the screen until the operator takes action, flush all streams, close all streams, and terminate execution. It should be used whenever an error has been identified and the programmer does not know how to continue meaningful execution. A call on `fatal()` is like a call on `printf()`, with a format and a list of variables to print.

**Fourth box: setting the stream format.** If the file opening was successful, we come to the fourth box. We intend to output doubles, and we want the output to be in two neat columns like a table. To do this, we send two *stream manipulators* to the output stream. `endl` and `flush` are also stream manipulators.

- This is done outside the loop because it only needs to be done once.
- `fixed` gives us neat columns, where every double or float is output with the same number of decimal places.
- `setprecision(4)`, when used with `fixed`, specifies that 4 digits should be printed after the decimal point.

**Fifth box: writing the table to the output file.**

- When the primary output of a program goes to a file, it is important to display comments on the screen so that the user knows the program is functioning properly. In this program, This is the only output that will be displayed on the screen.
- Each pass through the loop writes one line of output into the file. Because the step size is a non-integer amount, the loop termination test incorporates a fuzz factor to make sure the final value of `t=1` is processed.
- Inside the loop, the program prints a line for each value of `t`. Since the ouput is two numbers, the code must explicitly output space between them, in this case it is a tab character.
- The first and last few lines of output from this program follow.

```
0.0000 0.0000
0.0667 3.5882
0.1333 5.8889
...
0.9333 9.9802
1.0000 9.9873
```

**Last box:** *cleaning up.* As a matter of good style, the program closes the stream, and therefore the file, as soon as it is done writing it. If the programmer fails to close a file, the system will close it automatically when the program exits.

## 14.4 Stream Input

### 14.4.1 Input Functions

Previously, we have used `cin >>` to read keyboard input into variables. Here we introduce additional C++ functions for reading data. There are many variations on the ways that these functions can be used; only the most basic are listed here.

- `streamName >> var1 >> var2 ...;` formatted input.

The method for reading formatted input from a programmer-defined stream is the same as reading from `cin`. Use this form for reading all primitive types of data: character, integer, float, double, pointer. Multiple values can be read on one line by chaining the `>>` operators.

When the read operation starts, leading whitespace characters (newline, tab, space) are skipped. Then the visible characters in the stream are read, possibly converted to numbers, and stored in `var1`. The

Compute voltage as a function of time for the circuit in Figure 14.7.

```
#include "tools.hpp"
#define OUTFILE "voltage.dat"

#define V 10           // battery voltage
#define TAU 0.15       // time constant for this circuit
#define step (1.0 / 15.0) // time step for lines of table
#define epsilon (step / 2.0) // for floating comparison to end loop

double f( double t ) { return V * (1 - exp( -t / TAU )); }

int main( void )
{
    double t;           // time since switch closed
    double v;           // voltage at time t

    ofstream voltFile( OUTFILE );
    if (!voltFile.is_open()) fatal( "Cannot open output file: ", OUTFILE );

    voltFile <<fixed <<setprecision(4);

    cout << "\n Writing data to file " <<OUTFILE <<"\n\n";
    for (t = 0.0; t <= (1.0 + epsilon); t += step ) {
        v = f( t );
        voltFile <<t <<" " << v <<endl;
    }

    voltFile.close();

    return 0;
}
```

Figure 14.8. Writing a file of voltages.

declared type of `var1` determines whether a char or a number is read. If the instruction has more than one `>>` operator, whitespace is skipped again, and the next visible field is read into `var2`, etc.

The input operator can also be used to read a c-string into an array. It skips leading whitespace and reads characters until the next whitespace char. Thus, it reads a single word. You cannot read more than one word at a time with `>>`. There is a serious problem with this operation: there is no way to limit the length of the read to the available memory space. If the word is longer than the array it is stored in, the excess characters will be stored anyway. They will overwrite whatever variable is next in memory. For this reason, it is unprofessional to use `>>` to read string input

- `getline( streamName, stringName );` unformatted input.

Read a line from the file into a string. Enlarge the string, if necessary, to hold the whole line, and put a null terminator on the end. Do not skip leading whitespace. This is the easiest and safest way to read a line of text.

- `streamName.getline( arrayName, limit );` unformatted input.

Read characters from a file into the array until `limit-1` characters have been read and stored, or until a newline is found. Put a null terminator on the end. Do not skip leading whitespace; reads whatever is there, character by character, and store it in the array. The newline character is removed from the stream and not stored in the buffer.

- `streamName.get( charVariable ) ;` unformatted input.

Use this form for reading a single character if you *do not* want leading whitespace to be skipped.

**Skipping whitespace.** The bullet points above describe the default behavior of the functions. That can be changed and controlled using stream manipulators:

<code>hex and dec</code>	Put the stream in hexadecimal or decimal mode.
<code>ws</code>	Skip all whitespace chars in the stream; end at the next visible character.
<code>skipws</code>	Skip leading whitespace when doing formatted input.
<code>noskipws</code>	Stop skipping leading whitespace and read every character.

For example, to read a character without skipping leading whitespace, you might write: `cin >> noskipws >> charVariable;`

#### 14.4.2 Detecting End of File

The program examples so far have relied on the user to enter data at the keyboard. This works well when the amount of data is minimal or each input depends on the results of previous inputs. It does not work well for programs that must process large masses of data or data that have been previously collected and written into a file for later analysis. For these purposes, we need file input. File input raises a new set of problems that require new solution techniques. These problems include recognizing the end of the data and handling input errors.

When you are using keyboard input, running out of data is usually not an issue. Interactive programs typically provide an explicit way for the user to indicate when to quit. Until the quit signal is given, the system will continue to prompt for input and simply wait, forever if necessary, until the user types something. However, when your data come from a file, the file *does* have a definite end, and the end of data must be recognized as the signal to quit.

**Stream status flags.** Every C++ stream has a set of three status flags: `bad`, `fail`, and `eof`. These are set to `false` when a stream is opened. The `eof` flag is set to `true` when end of file is recognized. The `fail` flag is set to `true` when there is a conflict between the contents of a stream and the type of the variable that is supposed to receive the input. `bad` is set to `true` when any other kind of error happens while reading.

The simple way to check for the end-of-file condition is to use the stream function `eof()` to read the status flag in the stream. The next section gives an example of a properly used end-of-file test.

The function `good()`; returns `true` for a read with no problems and `false` when `eof` or any kind of error happens. It does not distinguish between end of file and errors. Some people use `good()` instead of `eof()` to test for end of file. This practice conflates errors and `eof`, and causes a program to treat errors the same way a normal end of file is treated, which can be a bad idea.

**Detecting eof for unformatted input.** Remember that unformatted input just reads a stream of raw characters, while formatted input performs input conversions. A failed input conversion will set the `fail` flag in the stream. However, an unformatted input operation cannot set that flag. Thus, using `good()` to test for end of file makes sense with `getline` but not with `>>`.

The result returned after calling `getline` is an `istream&`. You can write a short program to prove this: if you use the return value to initialize a local `istream&` variable, then you can use the variable for testing the stream's status flags. However, this is not something that anyone would normally do. What is done is to use that return value to break out of a read loop. This works to read all the lines of a text file and end when eof happens:

```
while (getline( mystream, mybuffer )) {... process the data.}
```

It works because the `istream` class defines a type conversion from type `istream&` to type `bool`. If all the status bits are false, the `bool` will be `true`. If `eof` or `fail` or `bad` is true, the `bool` will be `false`, causing control to leave the loop. This is illustrated in the code example in Figure 14.10.

### 14.4.3 Reading Data from a File

Once an input stream has been properly opened, data can be read from it. The syntax for reading from any `istream` is the same as the syntax for using `cin`. This is illustrated in the next program, Figure 14.9, which incorporates two important techniques: opening a file whose name is entered from the keyboard and handling an end-of-file condition.

It often is necessary to retrieve analytical or experimental data stored in a file. These values usually are analyzed or processed in some manner. However, in this example, we just send them to the screen. For the sake of this program, we assume that the file consists of pairs of numbers in the output format generated

This program prompts the user to enter the name of a data file. This offers more flexibility than simply writing the file name as a literal constant in the program since it permits the program to be used with a variety of input sources. However, it introduces a new opportunity for error: the user could give an incorrect name.

#### Notes on Figure 14.9. Reading a data file.

##### *First box: Header files.*

- This text uses an accompanying library of useful commands and functions named `tools`. The tools header file, `"tools.h"` #includes all of the standard C++ header files that are used in the text, as well as the necessary context declaration: `using namespace std;`.
- The first tools function that we use is `fatal()`, which gives an easy and succinct way to handle a serious error and display essential information about the error.

##### *Second box: Declarations.*

- Since the name of the file will be entered from the keyboard, we need to declare a string variable to hold whatever file or path name the user enters.
- We also declare a `stream` named `input` for the data file.

##### *Third box: Read in the file name.*

- To avoid all possibility that the read operation might overflow memory, we use a C++ string, not a char array. The string will be lengthened to hold whatever is entered, no matter how long.
- The program prompts the user and reads the name of a file using the form of `getline` that works with strings. If the file exists in the current default directory, the user needs to type only its name. However, if it is somewhere else in the file system, the user must supply either a complete pathname or a pathname relative to the current directory.

##### *Fourth box: Open the stream and check for errors.*

- We use the `open()` function instead of opening the file in the declaration because the file name was input at run time.

---

Read and process a file that has two numbers on each line in point coordinate format ( $x, y$ ).

```
#include "tools.hpp"      // for ios, iostream, fstream, iomanip, string, namespace

int main( void )
{
    double t, v;
    string fileName;      // File name or pathname of the file to read.
    ifstream input;

    // Create stream and open file. -----
    cout <<"What file do you want to read? ";
    getline( cin, fileName );           // Read entire line.

    input.open( fileName );
    if (!input.is_open())
        fatal( "Cannot open input file named ", fileName.c_str() );

    for (;;) {
        input >>t >>v;            // Read next input. -----
        if (input.eof()) break;      // Test stream status. ---
        cout <<t <<" " <<v <<endl;   // Echo input. -----
        // Process the input here, if processing is needed. -----
    }
    fclose( input );
    return 0;
}
```

---

**Figure 14.9.** Reading a data file.

- The program opens the stream `input` for reading whatever file the user named. A run-time error will occur if this file does not exist, if a pathname is given incorrectly, or if the user lacks permission to read it from the specified file or directory. All these conditions will be indicated by a `nullptr` value stored in `input`. The user will see this dialog:

```
What file do you want to read? junk
Cannot open input file named junk
```

- The fatal function cannot handle C++ strings, so we call the function `c_str()`, in the `string` class, to extract the C string from the C++ string.

**Third box: reading and echoing the data.**

- Each pass through the loop reads and echoes one line in the file. Further processing could be inserted in the indicated position.
- Line 2 of the loop reads the two data values from the file, converts them to type `double`, and stores them in `t` and `v`.
- Line 4 of the loop echo-prints the two values. It is always a good idea to echo the data, but it is especially so here because there is very little other output.

**Inner box: End-of-file handling.**

- The function `eof()` tests a flag set by the system when the program attempts to read data that are not there. This flag is *not* set when the last actual data value is read. To trigger an end-of-file condition, the program must try to read beyond the end of the file.

- That means that the eof test cannot be on the first line of the loop. So an `if ...break` after the read operation is used to end the loop.
- To be valid, the `eof()` test should be made after each line of data is read.
- An alternative that works is to call `if (! input.good()) break;`  
The function `good()` returns true if all input requests have been satisfied and there is good data to process.
- Here is some sample output, run on the file `voltage.dat` written by Figure 14.8 (dashes indicate lines that have been omitted):

```
What file do you want to read? voltage.dat
 0.00      0.00
 0.07      3.59
 0.13      5.89
-----
 0.80      9.95
 0.87      9.97
 0.93      9.98
 1.00      9.99
```

#### 14.4.4 Using Unformatted I/O

The preceding two programs used formatted I/O because they were processing numbers. Sometimes a program processes only non-numeric text that can be read and processed one line at a time. The next example shows a simple way to handle error detection and end of file. This works for string input, but not for any other type of input. Specifically, it does not work for a character array, a single char, or for numbers.

##### Notes on Figure 14.10. Using unformatted input.

###### *First box: Declarations.*

- We need a string variable to store the name of the input file, which will be read from the keyboard.
- We need a string variable to store the name of the output file, which will be created by concatenating the string ".copy" to the end of the input file name.
- We need a string variable to store the input data, which will be read and processed one line at a time.

###### *Second box: File names.*

- Use a C++ string for input any time you are not sure how many characters that input will be. We use unformatted input here to read a file name from the keyboard into the string `inName`.
- The + operator here is string concatenation. A new string is created by copying the letters in `inName` followed by the letters in ".copy". The string `inName` is not modified.

###### *Third box: Opening the files.*

- There is nothing new here. This code is the same as the previous example.
- Remember that every stream you open must be tested immediately. Although it would be possible to test both streams in one if statement, that is not a good idea. It is important to provide a high-quality error comment for the user – one that includes the word input or output and gives the name of the file that was not opened.

###### *Fourth box: Using a while loop.*

- Because of the way C++ handles the end of a file, it is normally not advisable to combine the input action and the test for eof. The reason is that the eof flag is not turned on until you attempt to read something that is not there. When that happens, the former contents of the input variables are not changed. So if the program goes on to process those values in the body of the loop, the last line will be processed twice. The cure is to use the infinite-for loop to do the repetition and an if...break to test for eof and leave the loop. This was shown in Figure 14.9.

- But there is another possibility when using unformatted input. The call on `getline()` returns a reference to `inStr`. When you use this reference in an `if` or `while` statement, it is converted to `false` if an error or `eof` happened during the read, and `true` if all is well. So the loop ends when `eof` happens, the `eof` bit gets set in the stream. Then when the `istream&` is converted to a bool, it is `false`.

#### 14.4.5 Reading an Entire File at Once.

There are multiple variations of `getline()`. One useful variation allows us to read a small or very large chunk of data from a file in a single read operation, stopping at the first occurrence of a *sentinel character*. This is useful for parsing ordinary input, but can also be used to read an entire file.

- Choose any character (preferably a visible character) that does not occur in the file.
- Add it to the end of the file. This is the sentinel character.
- Use it as the third parameter in a call on `getline()`
- The entire file (excluding the sentinel) will be read and stored in the input buffer.

**Notes on Figure 14.11. Reading an entire file.** Everything is the same as the previous program except for the two boxed portions.

---

Copy a text file one line at a time. Text files normally end with a newline character. In this program, the output will end with a newline whether or not the input did.

```
#include "tools.hpp"           // Include <iostream>, <iostream>, <fstream>, <string>
int main( void )
{
    string inName, outName, buffer;
    ifstream inStr;
    ofstream outStr;
    cout << " What file do you want to copy?  ";
    getline( cin, inName );        // Read entire line.
    outName = inName + ".copy";   // Make a related output file name.

    inStr.open( inName );
    if (!inStr.is_open()) fatal( "Can't open input file: %s", inName.c_str() );
    outStr.open( outName );
    if (!outStr.is_open()) fatal( "Can't open output file: %s", outName.c_str() );

    cout << " Ready to copy the file.\n";
    // Read and write one line at a time.
    while (getline( inStr, buffer)) {  // Loop will exit on end of file.
        if (inStr.good()) outStr << buffer << endl;
        else fatal( "Error reading input stream; aborting." );
    }
    cout << " Done.\n\n";
    inStr.close();
    outStr.close();
    return 0;
}
```

---

Figure 14.10. Copying a file.

---

Copy a text file in one operation.

```
#include "tools.hpp"           // Include <iostream>, <fstream>, <string>
int main( void )
{
    string inName, outName, buffer;
    ifstream inStr;
    ofstream outStr;

    cout << " What file do you want to copy? ";
    getline( cin, inName );      // Read entire line.
    outName = inName + ".copy"; // Make a related output file name.

    inStr.open( inName );
    if (!inStr.is_open()) fatal( " Can't open input file: %s", inName.c_str() );
    outStr.open( outName );
    if (!outStr.is_open()) fatal( " Can't open output file: %s", outName.c_str() );
    cout << " Ready to copy the file.\n";
    // Read entire file and copy it.

    getline( inStr, buffer, '#' ); // Read entire file, which ends in a #.

    if (inStr.good()) outStr << buffer << endl;
    else fatal( " Error reading input stream; aborting." );

    cout << " Done.\n\n";
    inStr.close();
    outStr.close();
    return 0;
}
```

---

Figure 14.11. Reading an entire file.

***First box: the call on getline().***

- The third parameter in this function call is the sentinel character. Everything up to the first occurrence of # will be read and stored in the string named **buffer**.
- The # will be removed from the stream but not stored in the buffer.

***Second box: the error test and output.***

- Always check whether your input operation worked correctly! In this case, if it does, we write the contents of **buffer** to the output file.
- If there was an error or an unexpected eof, we abort with an error comment.
- This program works properly whether or not there is a newline character on the last line.
- However, if the # is missing, the program aborts with an error comment.

## 14.5 Stream Errors

Files that are supposed to exist but seem not to, disk errors during reading, input conversion errors, and end-of-file conditions are situations that might occur. A **robust program** anticipates these things, checks for them, applies an appropriate recovery strategy, and continues processing if possible. Problems that cannot be handled routinely should be **flagged**; that is, the operator should be notified of the nature of the problem and the specific data that caused it. In this section and the next two, we look closely at **error detection** and how we may protect programs against some errors.

### 14.5.1 A Missing Newline

The paradigm for reading input from a file, presented in Figure 14.9, is simple and works reliably as long as the data file contains appropriate data. However, three kinds of errors in the data file will cause this program to malfunction.

**A missing newline.** If the newline character at the end of the last line of data is missing, the end-of-file flag will be turned on when the last item is read successfully. This is one read operation sooner than expected. The loop will test the flag before processing the last data set and the program will end immediately without processing the data. The result is shown here—the last line is missing:

```
What file do you want to read?  volts2.dat
 0.00      0.00
 0.07      3.59
 0.13      5.89
-----
 0.87      9.97
 0.93      9.98
```

But this is not a programming error; it is a data file error. It *is* normal to end each line of a file with a newline character. The program is “correct” because it handles a correct data file properly.

It is possible to write the input loop so that it does the “right” thing, whether or not, the final newline is present. This kind of error handling is discussed in the next section.

### 14.5.2 An Illegal Input Character

Even worse behavior will happen if the format calls for a numeric input and the file contains a nonnumeric character. The character causes a conversion error, which sets the fail bit in the input stream. Control will return immediately, even if the program statement calls for more input. Values read before the conversion error will be stored correctly in their variables, but the variable being processed when the conversion error happened will not be changed and still will contain the data from the prior line of input. Finally, the stream cannot be used again until its error flags are reset.

We modified an input file named `volts2.dat` by changing the number `0.1333` on the third line by using letter instead of digits for 0 and 1. The result is subtly different and easy to miss: `0.1333`. Then we ran the program from Figure 14.9 on the modified data. The results follow:

```
What file do you want to read?  voltsbad.dat
 0.00      0.00
 0.07      3.59
 0.07      3.59
 ....      ....
```

Even though the program tried to read line 4, nothing happened. The output went on and on, printing garbage, until the CTRL-C terminated it.

**Error recovery.** To read more input from the stream, two things must be done:

- Clear the error flag by calling `streamName.clear()`.
- Remove the offending character from the stream. This can be done by calling `char ch = get();`, or by using the `streamName.ignore(1)` to remove one character.

If the program is using keyboard input, it is desirable, after an error, to clear out any keystrokes that remain in the input buffer. This can be done with `streamName.ignore(255)`, which will clear out the entire line, up to the newline character, if the line is less than 255 characters long. Finally, the tools library defines an istream manipulator named `>> flush` that is very useful with keyboard input. It skips everything that is currently in the input buffer and leaves the stream empty.

We modified the read loop from the program from Figure 14.9 to add error detection and recovery code:

```

for (;;) {
    input >>t >>v;                                // Read next input. -----
    if (input.good()) cout <<t <<"      " <<v <<endl;
    else if (input.eof()) break;                  // Test stream status. ---
    else {
        input.clear();
        input.ignore(1);
    }
}

```

The results of a run using the same modified file are:

```

What file do you want to read? voltsbad.dat
0.00      0.00
0.07      3.59
333.00    5.89
0.20      7.36
0.27      8.31
....      ....

```

You can see that, in recovering from two consecutive errors, the resulting output became garbage. However, the program did run to completion and display enough information to permit a human to find and correct the error in the file.

Then we made a change at the end of the line instead of at the beginning: 0.1333 5.8SSS.

This is a different situation because, by the time the error happens, the system has read what seems to be two reasonable numbers. When the first ‘S’ is read, it stops the input operation and the number 5.8 is stored in v. The stream state is still good and the input is printed. Note, though, that it is not quite what it is supposed to be. The next time around the loop, a non-digit is read and it triggers a conversion error. We clear and ignore 1. This is repeated two more times. Finally the beginning of the 4th line arrives, with numbers, and the program continues normally. The results are:

```

What file do you want to read? voltsbad.dat
0.00      0.00
0.07      3.59
0.13      5.80
0.20      7.36
....      ....

```

**Data omission.** Similar errors can occur if some portion of a multipart data set is missing. In some cases, this will be detected when the next input character is the wrong type. For instance, if a letter was read but the program expected a digit, the operation would fail. In simpler file formats, such as lists of numbers separated by whitespace, the program would not detect an error, but a misalignment in data groupings would cause the output of the program to be meaningless.

## 14.6 File Application: Random Selection Without Replacement

The need to randomize the order of a list of items arises in certain game programs; for example, those that shuffle an imaginary deck of cards and deal them. The same problem also arises in serious applications, such as the selection of data for an experiment where a randomly selected subset of data items must be drawn from a pool of possible items.

The automated quiz program presented here is a good example of the use of files, random selection, and an array of objects. Each question in the drill-and-practice quiz is represented as one object in an array. The program presents the questions to the user, checks the answers, and keeps score. It uses C++’s pseudo-random number generator<sup>4</sup> and a “shuffling” technique to randomize the order of the questions.

This application has a main program, a controller class, and a data class. The data class defines a data object, in this case, the relevant information about one chemical element. The controller class handles a collection of data objects, carries out the logic of a quiz game, and interfaces with `main()`.

---

<sup>4</sup>This was presented in Chapter 7, Figure 5.26.

**Problem scope:** Write a drill-and-practice program to help students memorize the symbols for the chemical elements.

**Input file:** The list of element names, symbols, and atomic numbers to be memorized. One element should be defined on each line of the file, with the number first, followed by the name and symbol, all delimited by blanks.

**Keyboard input:** When prompted with the element name, the user will type in an element's atomic number and symbol.

**Output:** The program will display the names of the elements in random order and wait for the user to type the corresponding number and symbol. After the last question, the number of correct responses will be printed.

**Limitations:** Restrict the program to 20 elements.

---

**Figure 14.12. Problem specifications: A quiz.**

Program specifications are given in Figure 14.12, the main program in Figure 14.13, and a call chart for the final solution is in Figure 14.14. The purpose of this particular quiz game is to help the user learn about the periodic table of elements. However, the data structures and user interface easily can be modified to cover any subject matter.

The complete program discussed here was developed using top-down design. The result is a highly modular design, as shown in the call chart of Figure 14.14. (Calls on `iostream` functions are omitted from the chart.)

---

This main program implements the specification in Figure 14.12. It uses the classes declared in Figures 14.15 and 14.16, and the functions defined in 14.17 and 14.19.

```
#include "tools.hpp"
#include "quiz.hpp"

int main( void )
{
    cout << "\n Chemical Element Quiz\n ";
    string fileName;
    ifstream fin;

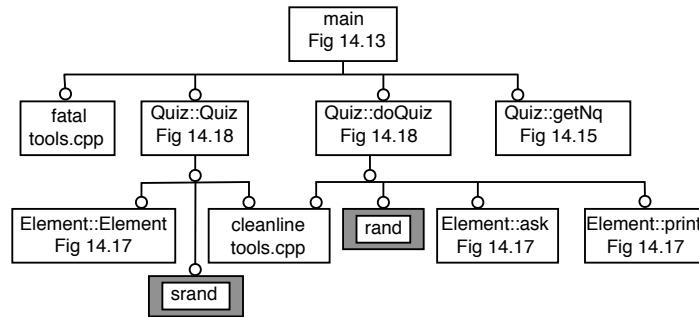
    cout << "\n What quiz file do you want to use?  ";
    getline( cin, fileName );
    fin.open( fileName );
    fatal( " Error:  cannot open input file: %s", fileName.c_str() );

    Quiz qz( fin );
    int correct = qz.doQuiz();
    cout << "\n You gave " << correct << " correct answers on "
        << qz.getNq() << " questions.\n ";

    fin.close();
    return 0;
}
```

---

**Figure 14.13. An elemental quiz.**



**Figure 14.14. Call chart for the Element Quiz**

#### Notes on Figure 14.13: An elemental quiz.

- The main program prints opening and closing messages. Both are important to keep the human user informed about the state of the computer system.
- It both opens and closes the data file. These two actions should both be in `main` or be in the constructor and destructor of the same class.

#### *First and third boxes: local declarations.*

- Most local variables should be declared at the top of a function where they can be easily found. However, sometimes it is wise or even necessary to wait until the program has the data needed to initialize the variable.
- There are three local variables in this function: a string for the filename that the user will enter, a stream to bring in the data, and a `Quiz` object. The stream and the string are declared in Box 1, at the top.
- The `Quiz` is declared later, in Box 3 because the `Quiz` constructor calls for a stream parameter, and the stream is opened in the Box 2.

#### *Second box: opening the input stream.*

- The data file can be opened here or in the `Quiz` class. We open it in `main` to demonstrate how to pass an open stream to a class constructor.
- `main` carries out the usual opening process: (a) prompt for a file name, (b) open the input file, (c) test to be sure it is open, and (d) call `fatal()` if it cannot be opened. An error output would look like this:

Chemical Element Quiz

```
What quiz file do you want to use? q1
Error: cannot open input file: q1
```

#### *Third box: the primary object.*

- In an OO program, the job of the `main()` function is to create the first object, then transfer control to it by calling one of its functions.
- The first line here creates a `Quiz` object that will use the stream that was just opened.
- The second line uses the new `Quiz` object to call the `doQuiz()` function, which does the work.
- The return value from `doQuiz()` is used as part of the program's closing comment.

#### Notes on Figure 14.15. The Quiz class.

This is a controller class. Only one `Quiz` object will be created by `main`. It will carry out the logic of the quiz game.

#### *First box: data members.*

- A controller class usually manages an input stream and often also an output stream. Sometimes it opens both streams (and later closes them). Sometimes the streams are opened and closed in `main()`.
- A controller normally has one or more collections of data objects. In this case, an array of `Elements`.
- The remaining data members are used to manage the game or business logic. In this case, we have integers to store the number of questions in the quiz and the number that the user answered correctly.

---

This file declares the Quiz class and is included by Figures 14.13 and 14.17.

```
#include "tools.hpp"                                // File: quiz.hpp
#include "elem.hpp"
// -----
#define MAXQ 20           // Maximum number of questions in the quiz
class Quiz {
private:
    istream& fin;
    Element elem[MAXQ];
    int nq = 0;        // Number of questions in the quiz.
    int score = 0;     // Number of correct answers.

public:
    Quiz( istream& openFile );
    int doQuiz();
    int getNq(){ return nq; }
};


```

---

**Figure 14.15.** The Quiz class declaration.

- They are initialized here in the class declaration because we know the correct initializations and because it makes writing the constructor easier to do it here.

***Second box: function members.***

- Prototypes and one-line functions are given in the class declaration.
- Every class has one constructor, often two. The responsibility of a constructor is to initialize the data members of the class so that the class is ready to use. This often includes reading in a file of data.
- In a controller class, the constructor is called from `main()`. When construction is finished, `main()` calls the controller's primary work function, in this case named `doQuiz()`. Control returns from `doQuiz()` to `main()` at the end of execution, and `main()` cleans up and terminates.
- Because these two functions are long, they are defined in the implementation file for Quiz, not here in the header.
- The third function here is a one-line getter. It gives read-only access to a private data member of the Quiz object.

**Notes on Figure 14.16. The Element class.** This header class is included by the main program in Figure 14.13 and by the Element implementation file in 14.17.

***First box: data members.***

- This is a data class: it has data members that represent the significant properties of a real-world object, and functions that facilitate the use of the data object.
- From the program specification, we see that three data members are needed.
- None of these members are initialized in the class declaration because all of them must be read from the input file at run time.
- We use strings (not char arrays) because we need to input the element name and symbol. Input with `string` is easy. Input with `char[]` is not easy and has been the cause of many security vulnerabilities. The current security recommendation is to use type `string` any time a variable is used for input.
- Type `short` is used instead of `int` because we know that atomic numbers will not exceed 3 digits.

---

This header file is included by the main program in Figure 14.13 and the class implementation in Figure 14.19.

```
#include "tools.hpp"                                // File: elem.hpp

class Element {
private:
    string name;           // name of element.
    string symbol;          // symbol, from periodic table.
    short number;           // atomic number.

public:
    Element() = default;
    Element( string nm, string sy, short at );

    bool ask();
    void print( ostream& out );
};


```

---

**Figure 14.16.** The Element class declaration.

**Second box: constructors.**

- The first constructor is a default constructor. It initializes the whole object to 0 bits. We need a default constructor here because the client class, Quiz, creates an array of Elements. To create an array of a class type you must provide a default constructor.
- The second constructor is the one that will be used to create actual Element objects. It receives three pieces of data as parameters and stores them in the corresponding three data members.

**Third box: function members.**

- Design principle: a class should protect its members.  
Design principle: a class should be the expert on its own members.
- The data members of Element are private (principle 1). The functions defined here, in the public area, must provide all appropriate ways for Quiz (the client class) to use Elements. A client class should *not* be reaching down into an Element object to handle its parts.
- The `ask()` function is called from `doQuiz()` each time a new random question is selected. Then `ask()` carries out the interaction with the user, prompts for answers, corrects wrong answers, and returns a true/false result when the answer was right/wrong.
- Every class needs a print function. `Element::print()` is called from `doQuiz()` each time a users answer is wrong. It formats the element data nicely and sends it to whatever output stream is specified by the parameter.

However, even if the client did not need to print the data, the class should still have a print function. During the construction and debugging phases, it is needed.

**Notes on Figure 14.17. Implementation of the Element class.** This is a data class: it has data members that represent the significant properties of a real-world object, and functions that facilitate the use of the data object.

**First function: the constructor.** This is the simplest possible kind of constructor. It takes each parameter and stores it in the corresponding data members. Every member has a corresponding parameter.

---

These functions are called from the main program in Figure 14.13. They rely on the header files declared in Figures 14.15 and 14.16.

```
#include "elem.hpp"                                     File: elem.cpp
// -----
Element:: Element( string nm, string sy, short at ) {
    name = nm;
    symbol = sy;
    number = at;
}

// -----
bool Element:: ask() {
    string symb;                                // User response for atomic symbol.
    short num;                                   // User response for atomic number.
    cout <<"\ Element: " <<name <<"\n";
    cout <<"\t symbol ? ";
    cin >> ws;
    getline( cin, symb );
    cout <<"\t atomic number ? ";
    cin >> num;
    return (symbol == symb && number == num );
}

// -----
void Element:: print( ostream& out ) {
    out <<name <<" is " <<symbol <<": atomic number " <<number;
}
```

---

**Figure 14.17.** The implementation of the `Element` class.

***Second function: ask().***

- This is an ordinary input function: prompt for a series of inputs, read each one, and store it in a class data member.
- One of the inputs is a string. To read it, we use `string-getline`. But `getline()` does not skip leading whitespace and cannot be combined with `>>` in an input chain. Therefore, getting the data requires two lines of code: `cin >> ws; getline( cin, symb );` The other input is a number and can be read simply.
- The return statement compares two strings. Because they are C++ strings, we can use `==`. If they were C-strings, we would have to use `strcmp` instead.
- The last line returns the result of the `&&` operation. No if statement is needed; don't use one.

**Notes on Figure 14.18. The Quiz constructor.**

***First box: the function header and ctor initializer.***

- The job of a constructor is to initialize the new class object so that it is fully functional and ready to use. For `Quiz`, this means reading an input file.
- The input file was opened in `main()` and passed as a parameter to the constructor. The type of the parameter is `istream&`. This presents a new situation and requires a new syntax.
- One of the `Quiz` class members is an `istream&`. One of the properties of a `&` variable is that you must use initialization, not assignment, to give it a value. A `ctor` is a syntax for initializing a class member.

These functions are called from the main program in Figure 14.13. They rely on the header files declared in Figures 14.15 and 14.16.

```
#include "quiz.hpp"
// -----
Quiz::Quiz( istream& openFile ) : fin(openFile) {
    short number;
    string name, symbol;
    for (nq = 0; nq < MAXQ; ) {
        fin >> name >> symbol >> number;
        if (fin.good()) {
            elem[nq] = Element( name, symbol, number );
            ++nq; // all is well -- count the item.
        }
        else if (fin.eof()) break;
        else {
            fin.clear();
            cerr << " Bad data while reading slot " << nq << endl;
            cleanline(fin); // Skip remaining characters on the line.
        }
    }
    srand( time( NULL ) ); // initialize random number generator.
}
```

**Figure 14.18. The Quiz Constructor.**

- Ctors are written after the end of the parameter list in a constructor and before the opening { for the class.
- To write a ctor, write a colon followed by the name of a class member, followed by the initializing expression in parentheses. In this case, the initializing expression is the name of the parameter.
- In general, ctors are used in constructors anywhere that something must be initialized, not assigned. An example is a `const` data member.

#### *Second box: the input loop.*

- This function reads a series of data lines from the designated quiz input file. Each line of data is used to construct an Element, and that element. is stored in an array of Elements, which is a data member of the class.
- There is an upper limit, `MAXQ` on the number of questions in a quiz. However, any particular file could be shorter, and there might be unreadable lines in the file. For this reason, we have a class member, `nq` that stores the actual number of well-formed Elements that have been stored in the array `elem`: `nq <= MAXQ`

#### *First inner box: the input.*

- In this box, we take tree inputs from the stream `fin`, then test the stream state. If it is still `good()`, we have successfully read three fields so we use them to create an Element.
- When we store an Element in the array, we increment the subscript. This is not done in the first line of the loop because some input lines might be bad. We want to count and store only the good lines.

#### *Second inner box: error handling.*

- If the stream state is not good, it might be because of an end of file, or it might be caused by a read error or a conversion error. The two situations require different handling.
- For end of file, we simply want to leave the loop. We have created `nq` Elements and stored them in the array. All is well.

- If any kind of error happened, we must clean up the mess. This is a 3-step process:
  - First, the stream error flags must be cleared: `fin.clear();`
  - We must inform the user about the error. The error stream is used for this purpose.
  - Finally, any part of the faulty line that is still in the stream must be removed to clear the way for reading the next correct item. The `cleanline()` function in the tools library does this job, up to the next newline character.
  - The following array contents would be the result of reading data from the file `quizz2.txt`, containing six data sets, into an array with 20 slots. We use these data for the remainder of the discussion. Figures 14.20 through 14.23 show how the array contents change during the quiz.

	<code>elem.name</code>	<code>.symbol</code>	<code>.number</code>
[0]	Sodium	Na	11
[1]	Magnesium	Mg	12
[2]	Aluminum	Al	13
[3]	Silicon	Si	14
[4]	Phosphorus	P	15
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
	...	...	...
[19]	??	??	?

**Last box: random numbers.** A constructor is supposed to initialize everything the class needs to function. Sometimes this goes beyond initializing the data members. This is an example. Games and quizzes rely on random numbers, and the system's random number function needs to be initialized. Here we call `srand()`, which "seeds" the random number generator.

#### Notes on Figure 14.19: The `doQuiz()` function.

##### *Outer box: Asking all the quiz questions.*

- `many` is set initially to the number of Elements in the array. Each time a question is used, `many` is decremented, so it is a true count of the remaining questions. We need this number for choosing a random question.
- This large loop asks all the questions, scores the answers, keeps score, and performs housekeeping on the array of questions. Finally it returns the score to `main()`. This would not be necessary, since the score is also stored as a class member. However, it is a convenience. Many system functions use more than one path to return an answer so that the programmer can use whatever is easiest.

##### *First inner box: Selecting a random question.*

- Each time around the loop there are fewer unasked questions. The strategy here is to keep the unasked questions at the low-subscript end of the array. (See notes on fourth inner box.)
- `rand()` returns a number between 0 and MAXINT. We want a number between 0 and `many-1` (the number of remaining questions). The mod operator gives this to us.

##### *Second inner box: Posing the question.*

- Design principle: Delegate the job to the expert.

When a question has been selected, we need to present it to the user. The Element class is the expert on all things relating to elements, including how to ask quiz questions about them. So we delegate this task to the expert by calling a function in the Element class, using the particular element we have selected: `elem[k].ask()`

- If `ask()` returns true we add 1 to the user's score and say something nice.

This function belongs in the implementation file for the Quiz class, after the Quiz constructor.

```

int Quiz::
doQuiz() {
    int many = nq;                      // Number of questions not yet asked.
    int k;                                // The random question that was selected.
    cout <<"\n This quiz has " <<nq <<" questions.\n"
        <<" When you see an element name, enter its symbol and number.\n";
    while (many > 0) {
        k = rand() % many;               // Choose subscript of next question.

        if( elem[k].ask() ){

            score++;
            cout<<"\t YES ! Score one.\n";
        }
        else {
            cout <<"\t Sorry, the answer is ";
            elem[k].print( cout );
            cout <<endl;
        }
        // Now remove the question from further consideration.
        --many;                           // Decrement the counter.
        elem[k] = elem[many];             // Move last question into vacant slot.
    }
    return score;
}

```

Figure 14.19. The doQuiz function.

*Third inner box: Always be polite.*

- If `ask()` returns false, we give a polite negative comment and show the correct information. After all, this IS about learning the periodic table!
- Following is the transcript of a question from one run of the program: the first three lines were written by `Element::ask()` and the last line by `Quiz::doQuiz()`, with part of the output delegated to `Element::print()`.

```

Element: Phosphorus
symbol ? Ps
atomic number ? 15
Sorry, the answer for Phosphorus is P: atomic number 15

```

- The Element class is the expert on showing information about elements. So we delegate the display job to Element by calling its `print()` function. The stream parameter could be a file-stream if we wanted file output.

*Last inner box: housekeeping.*

- Once a question has been used, we want to exclude it from future use. That question could be anywhere in the array, and we want to get rid of it.
- There is an easy trick for doing that. Decrement the loop counter, `many`, then take the element at subscript `many` and copy into the slot occupied by the element that was just used. We decrement first because `many` is the subscript of the first array slot that does not contain current good information.

First question is #4:

elem	.name	.symbol	.number
[0]	Sodium	Na	11
[1]	Magnesium	Mg	12
[2]	Aluminum	Al	13
[3]	Silicon	Si	14
[4]	Phosphorus	P	15
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
...		..	..
[19]	??	??	?

nq	many	k	score
8	8	4	0

question 1: Phosphorus

Before asking first question.

elem	.name	.symbol	.number
[0]	Sodium	Na	11
[1]	Magnesium	Mg	12
[2]	Aluminum	Al	13
[3]	Silicon	Si	14
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
...		..	..
[19]	??	??	?

nq	many	k	score
8	7	4	0

answer is wrong.

After asking first question.

Figure 14.20. The array after one question.

- This leaves assorted garbage in the end of the array, but that is OK because we never look at anything past many.

**Third box: remove the question from further consideration.**

- Figures ?? through ?? show the state of the question array after asking each of the first six questions of our sample quiz. The first three were answered correctly, but only one of the next three was right.
- After each question is asked, we decrement many, shortening the part of the array containing the unasked questions. Just after the decrement operation, many is the subscript of the last unused question. We copy this question's data into slot k, overwriting the data of the question that was just asked.
- In the diagrams, we represent the shortening by coloring the discarded slots gray. Note in Figure ?? that, after three questions are asked, three elements no longer appear in the white part of the array and all the unused questions have been shifted up so they do appear in the white area. The contents of the gray part of the array no longer matter; they will not be used in the future because the program always selects the next question from the white part (slots 0...many-1).
- Here is the sample dialog for questions 2 and 3, which were answered incorrectly:

```
Element: Magnesium
symbol ? Mg
atomic number ? 12
YES ! Score one.
```

```
Element: Silicon
symbol ? Si
atomic number ? 14
YES ! Score one.
```

- Note that, after the fifth question is asked in Figure 14.22, the question replacement operation effectively does nothing, since this is the last question in the list. We could have saved the effort of doing the copy operation, but a test to detect this special case would end up being more complex and more work in the long run than the unnecessary copying operation.
- The quiz continues until all questions have been presented. The last question has not been illustrated.

**Fourth box: The return.** When all the questions have been asked, doQuiz() returns the score to main(), which prints the results:

You gave 7 correct answers on 8 questions.

---

Second question is #1:

	elem .name	.symbol	.number
[0]	Sodium	Na	11
[1]	Chlorine	Cl	17
[2]	Aluminum	Al	13
[3]	Silicon	Si	14
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
...	..	..	
[19]	??	??	?

	nq	many	k	score
	8	6	1	1

question: Magnesium

After asking second question.

Third question is #3:

	elem .name	.symbol	.number
[0]	Sodium	Na	11
[1]	Chlorine	Cl	17
[2]	Aluminum	Al	13
[3]	Sulphur	S	16
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
...	..	..	
[19]	??	??	?

	nq	many	k	score
	8	5	3	2

question: Silicon

After asking third question.

Figure 14.21. The array after three questions.

## 14.7 What You Should Remember

### 14.7.1 Major Concepts

#### Streams and files.

- A file is a physical collection of information stored on a device such as a disk or tape. It has a name and its own properties. A stream is a data structure created and used during program execution to connect to and access a file or any other input source or output destination.
- There are three standard predefined streams—`cin`, `cout`, `cerr`, and `clog`—which are connected by default to the keyboard, monitor, and monitor, and a file, respectively. A programmer can define additional streams.
- Streams are declared to be input, output, or both. A stream can be opened in the declaration or by a later call on `open()`. Opening a stream clears its status flags, creates a buffer, and attaches the stream

---

Fourth question is #1:

	elem .name	.symbol	.number
[0]	Sodium	Na	11
[1]	Argon	A	18
[2]	Aluminum	Al	13
[3]	Sulphur	S	16
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
...	..	..	
[19]	??	??	?

	nq	many	k	score
	8	4	1	3

question: Chlorine

After asking fourth question.

Fifth question is #3:

	elem .name	.symbol	.number
[0]	Sodium	Na	11
[1]	Argon	A	18
[2]	Aluminum	Al	13
[3]	Sulphur	S	16
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
...	..	..	
[19]	??	??	?

	nq	many	k	score
	8	3	3	4

question: Sulphur

After asking fifth question.

Figure 14.22. The array after five questions.

---

Sixth question is #0:

elem .name	.symbol	.number
[0] Aluminum	A1	13
[1] Argon	A	18
[2] Aluminum	A1	13
[3] Sulphur	S	16
[4] Argon	A	18
[5] Sulphur	S	16
[6] Chlorine	Cl	17
[7] Argon	A	18
[8] ??	??	?
...	..	..
[19] ??	??	?

nq	many	k	score
8	2	0	5

question: Sodium

After asking Sixth question.

Seventh question is #0:

elem .name	.symbol	.number
[0] Argon	A	18
[1] Argon	A	18
[2] Aluminum	A1	13
[3] Silicon	Si	14
[4] Argon	A	18
[5] Sulphur	S	16
[6] Chlorine	Cl	17
[7] Argon	A	18
[8] ??	??	?
...	..	..
[19] ??	??	?

nq	many	k	score
8	1	0	6

question: Aluminum

After asking seventh question.

Figure 14.23. The array after seven questions.

to some device that will supply or receive the data. A call on `close()` terminates the connection and sets the stream status to not-ready.

- Once open, a stream can be used to transfer either ASCII text or data in binary format. All of the I/O functions presented in this chapter operate in text mode. Binary I/O is introduced in Chapter 15.
- Except for `cerr`, streams are buffered. On input, bytes do not come directly from a device into the program. Rather, a block of data is transferred from the device into a buffer and waits there until the program is ready for it. Typically, it is read by the program in many smaller pieces. On output, bytes that are written go first into a buffer and stay there until the buffer is full or flushed for some other reason, such as an explicit call on `flush()`. If a program crashes, everything waiting in the output buffers is lost. The corresponding files are not closed and, therefore, not readable.
- File-streams can be read and written using the same operators and functions that operate on the standard streams: `>>` and `<<`, `getline(streamName, stringName)`, `streamName.getline( char buffer[] )`, and `charVar = streamName.get()`.
- When reading data from a file, certain types of errors and exceptions are likely to occur:
  1. The `open()` function may fail to open the file properly.
  2. The end of the file will eventually be reached.
  3. An incorrect character may have been inserted into the file, polluting the data on that line.
  4. Data from a multipart data set may have been omitted, causing alignment errors.

Methods and functions exist to detect and recover from these types of errors. A robust input routine was presented that incorporates detection and recovery methods.

**New tools.** These definitions from the tools library are constantly useful. You should familiarize yourself with them:

- The function `fatal()` is used like `printf()`. Call it to write an error message and a file name to the standard error stream after a file-processing error. It flushes all stream buffers, closes all files, and aborts execution properly.
- The function `cleanline()` reads and discards characters from the designated stream until a newline character is encountered. It was used in the quiz program, Figure 14.18.

### 14.7.2 Programming Style

**Function follows form.** When creating a class for a program, its form must follow that of the real-world object the program is modeling. Within the program, processing of all data should be delegated to the class that defines that data. Operations should be performed by calling a function in that class. This keeps the details of the representation separate from the main flow of program logic.

Modules should be designed to permit the programmer to think and talk about the problem on a conceptual level, not at a detailed level. For example, the quiz program in Figure 14.13 operates on an array of objects. It uses functions to call up a quiz (operating on the whole array), administer a quiz (operating on the whole array), and ask a single question (operating on a single item).

**Opening files.** Failure to be able to open a file is a common experience. This leads to the following maxim: Always check that the result of `open()` is a valid stream. Further, if a program uses files, all of them should all be opened “up front,” even if they will not be used immediately. This avoids the situation in which human effort and computer time have been spent entering and processing data, only to find that it is impossible to open an output file to receive the results. The general principle is this: Acquire all necessary resources before committing human labor or other resources to a project.

**Closing streams.** All streams are closed automatically when the program exits. However, the `iostream` library provides the `close()` function to permit a stream to be closed before the end of the program. Closing an input file promptly releases it for use by another program. Closing an output file promptly protects it from loss of data due to possible program crashes later in execution. Closing a stream releases the memory space used by the buffers.

**Flushing.** The `flush()` function is defined in the C++ standard only for output streams. It is used automatically when a stream is closed. Normally, a programmer has no need to call it explicitly. The tools library defines `flush()` for input streams. This is helpful when an error occurs during interactive input.

**End of file.** The function `eof()` should be used to detect the end of an input file. This condition occurs when you attempt to read data that are not there, so test for end of file only *after* an input operation, not before it. This is the only way to distinguish normal `eof` condition from a stream error.

**Exceptions and errors.** File-based programs typically handle large amounts of prerecorded data. It is important for these programs to detect file input errors and inform the user of them. The input functions in the `iostream` library set error and status flags. These should be checked and appropriate action taken if it is important for a program to be robust. Also, an error log file can be kept. It should record as much information as possible about each input error, so that the user has some way to find the faulty data and correct them. The input routine developed in Figure ?? robustly handles many types of input errors. The order in which the error tests are made is important; use this example as a template in your programs.

### 14.7.3 Sticky Points and Common Errors

- Keep in mind the difference between a stream name and a file name. The file name identifies the data to the operating system. A stream name is used within a program. The `open()` function connects a stream to a file and the argument to `open()` is the only place the file name ever appears in a program.
- A common cause of disaster is to create an output file with the same name as an already existing file, which will “blow away” the earlier file. The most common “victims” of this error are the program’s own source code or input file, whose name absentmindedly is typed in by the user. For this reason, we recommend using something distinctive, such as `.in` as part of every input file name and `.out` as part of every output file name.
- Another common mistake is fail to skip whitespace explicitly when using `getline()`, or to expect `>>` to read more than one word.
- An input file must be constructed correctly for the program that reads it. Most programs expect the last line of a file to end with a newline character.
- The first unprocessed character in an input stream’s buffer often is the whitespace character that terminated the previous data item. Input formats must take this into account, especially when working with character or string input.

- Input errors, incorrectly formatted data, and end-of-file problems are frequently encountered exception conditions. Any program that is to be used by many people or for many months needs to test for and deal with these conditions robustly. Failure to do so can lead to diverse disasters, including machine crashes and erroneous program results.
- A nonnumeric character in a numeric field can throw a program into an infinite loop unless detected and dealt with. Minimally, the stream state must be cleared and a function such as `ignore(1)` must be called to remove the character that caused the error from the input stream. The function `cleanline()` removes the faulty character along with the rest of the current line.
- Use C++ strings when reading character data or entire lines of input. Do not use character arrays for this purpose. An array must be long enough to hold all of the characters that will be read, and that is totally unpredictable. If the array is too short, the extra characters will overlay something else in memory.

#### 14.7.4 New and Revisited Vocabulary

These are the most important terms and concepts discussed in this chapter:

stream I/O	log file	stream manipulator
buffer	status code	stream <code>getline()</code>
stream	error flag	string <code>getline()</code>
file	end of file	robust program
binary file	error detection	buffered stream
text file	error recovery	unbuffered stream

The following keywords, constants, and C++ library functions are discussed in this chapter:

<code>ios::</code>	<code>iomanip</code>	<code>get()</code>
<code>iostream</code>	<code>left and right</code>	<code>getline()</code>
<code>ostream</code>	<code>ws</code>	<code>&lt;&lt;</code>
<code>istream</code>	<code>scientific</code>	<code>&gt;&gt;</code>
<code>cin and cout</code>	<code>fixed</code>	<code>cleanline()</code>
<code>cerr</code>	<code>setprecision()</code>	<code>fatal()</code>
<code>clog</code>	<code>setfill()</code>	<code>flush()</code>
<code>open()</code>	<code>setw()</code>	<code>eof()</code>
<code>close()</code>	<code>hex and dec</code>	<code>good()</code>

#### 14.7.5 Where to Find More Information

- On the text website, a complete application for measuring torque is developed step-by-step.
- The file copy program in Figure 14.10 is reworked and simplified in Chapter 20. Interactive input is replaced by the use of command-line arguments.
- Binary-mode file input and output are also used in an image-processing application in Chapter 15.

# Chapter 15

## Calculating with Bits

Until now, we have used the basic numeric types and built complex data structures out of them. In this chapter, we look at numbers at a lower, more primitive level. We examine how bits are used to represent numbers and how those bits are interpreted in varied contexts. We present a variety of topics related to number representation and interpretation.

Most C and C++ programs use the familiar base-10 notation for both input and output. The programmers and users who work with these programs can forget that, inside the computer, all numbers are represented in **binary** (base 2), not **decimal** (base 10). Usually, we have no need to concern ourselves with the computer's internal representation.

Occasionally, though, a program must deal directly with hardware components or the actual pattern of bits stored in the memory. Applications such as cryptography, random number generators, and programs that control hardware switches and circuits may relate directly to the pattern of bits in a number, rather than to its numerical value. For these applications, base-10 notation is very inconvenient because it does not correspond in an easy way to the internal representation of numbers. It takes a little work, starting with a large base-10 number, to arrive at its binary representation. Throughout this chapter and its exercises, whenever a numerical value is used and it is ambiguous as to which number base is being used, that value will be subscripted. For example,  $109_{10}$  means 109 in base 10, while  $109_{16}$  means 109 in base 16.

Ordinary arithmetic operators are inadequate to work at the bit level because they deal with the values of numbers, not the patterns of bits. An entire additional set of operators is needed that operate on the bits themselves. In this chapter, we study the *hexadecimal notation*, (which is used with unsigned numbers), and the bitwise operators that C and C++ provides for low-level bit manipulation. Further details of the bit-level representation of signed and unsigned integers and algorithms for number-base conversion among bases 10, 2, and 16 are described in Appendix E.

Finally, we present *bitfield structures*, which provide a way to organize and give symbolic names to sets of bits, just as ordinary structures let us name sets of bytes.

### 15.1 Number Representation and Conversion

In this section, we explore the ways in which integers and floating-point numbers are represented in computers.

#### 15.1.1 Hexadecimal Notation

We use **hexadecimal** notation to write machine addresses and to interface with certain hardware devices. Using hexadecimal notation is the easiest way to do some jobs because it translates directly to binary notation<sup>1</sup>. A programmer who wants to create a certain pattern of bits in the computer and cares about the pattern itself, not just the number it represents, should first write out the pattern in binary, then convert it to hexadecimal, and write a literal in the program. To properly use the hexadecimal notation, we need to store values in **unsigned int** or **unsigned long** variables. The following paragraphs describe the syntax for writing hexadecimal (hex) literals and performing input and output with hexadecimal values.

---

<sup>1</sup>The conversion method is given in Appendix ?? and examples can be seen in Figures 15.12, 15.14, 15.15, 15.16, and 15.17.

---

An integer literal can be written in either decimal or hexadecimal notation:

Decimal	Hex	Decimal	Hex	Decimal	Hex
0	0x0	16	0x10	65	0x41
7	0x7	17	0x11	91	0x5b
8	0x8	18	0x12	127	0x7f
10	0xA	30	0x1E	128	0x80
11	0xB	33	0x21	255	0xFF
15	0xF	64	0x40	32767	0x7FFF

---

Figure 15.1. Hexadecimal numeric literals.

**Hexadecimal literals.** Any integer or character can be written as either a decimal literal (base 10) or a **hexadecimal literal** (base 16).<sup>2</sup> As illustrated in Figure 15.1, a hex literal starts with the characters 0x or 0X (digit zero, letter ex). Hex digits from 10 to 15 are written with the letters A...F or a...f. Upper- and lower-case letters are acceptable for both uses and mean the same thing. A **hex character literal** is written as an escape character followed by the letter x and the character's hexadecimal code from the ASCII table; a few samples are given in Figure 15.9. These numeric codes should be used only if no symbolic form exists because they may not be **portable**, that is, they depend on the particular character code of the local computer and may not work on another kind of system. In contrast, quoted characters are portable.

### 15.1.2 Number Systems and Number Representation

Numbers are written using positional base notation; each digit in a number has a value equal to that digit times a place value, which is a power (positive or negative) of the base value. For example, the *decimal* (base 10) place values are the powers of 10. From the decimal point going left, these are  $10^0 = 1$ ,  $10^1 = 10$ ,  $10^2 = 100$ ,  $10^3 = 1,000$ , and so forth. From the decimal point going right, these are  $10^{-1} = 0.1$ ,  $10^{-2} = 0.01$ ,  $10^{-3} = 0.001$ ,  $10^{-4} = 0.0001$ , and so forth. Figure 15.2 shows the place values for *binary* (base 2) and *hexadecimal* (base 16); the chart shows those places that are relevant for a short integer.

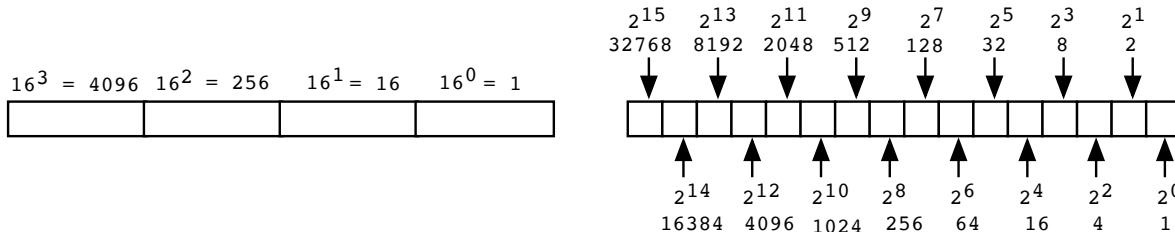
Just as base-10 notation (decimal) uses 10 digits to represent numbers, hexadecimal uses 16 digits. The first 10 digits are 0–9; the last six are the letters A–F. Figure 15.6 shows the decimal values of the 16 hexadecimal digits; Figure E.7 shows some equivalent values in decimal and hexadecimal.

---

<sup>2</sup>Octal literals can be used, too, but their use is not as prevalent and so they are omitted from this text.

---

Place values for base 16 (hexadecimal) are shown on the left; base-2 (binary) place values are on the right. Each hexadecimal digit occupies the same memory space as four binary bits because  $2^4 = 16$ .




---

Figure 15.2. Place values.

The binary representations of several signed and unsigned integers follow. Several of these values turn up frequently during debugging, so it is useful to be able to recognize them.

$\begin{array}{r} 32768 \\ = 16384 \\ = 8192 \\ = 4096 \\ = 2048 \\ = 1024 \\ = 512 \\ = 256 \\ = 128 \\ = 64 \\ = 32 \\ = 16 \\ = 8 \\ = 4 \\ = 2 \\ = 1 \end{array}$	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	0	1	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	Interpreted as a signed short int high order bit = -32768	Interpreted as an unsigned short int high order bit = 32768
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1																																																																																																																				
0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0																																																																																																																				
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1																																																																																																																				
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																																																																																																																				
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1																																																																																																																				
1	1	0	1	1	0	0	0	1	1	1	1	0	0	0	0																																																																																																																				
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																																																																																																																				
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1																																																																																																																				
		32767 10000 +1 0	32767 10000 +1 0																																																																																																																																
		-32768 + 32767 = -1 -32768 + 22768 = -10000 -32768 + 0 = -32768 -32768 + 1 = -32767	+32768 + 32767 = 65535 +32768 + 22768 = 55536 +32768 + 0 = 32768 +32768 + 1 = 32769																																																																																																																																

Figure 15.3. Two's complement representation of integers.

### 15.1.3 Signed and Unsigned Integers

On paper or in a computer, all the bits in an unsigned number represent part of the number itself. Not so with a signed number: One bit must be used to represent the sign. The usual way that we represent a signed number on paper is by putting a positive or negative sign in front of a value of a given magnitude. This representation, called *sign and magnitude*, was used in early computers and still is used today for floating-point numbers. However, a different representation, called *two's complement*, is used for signed integers in most modern computers.

In this notation, the leftmost bit position has a negative place value for signed integers and a positive value for unsigned integers. All the rest of the bit positions have positive place values. Numbers with a 0 in the high-order position have the same interpretation whether they are signed or unsigned. However, a number with a 1 in the high-order position is negative when interpreted as a signed integer and a very large positive number when interpreted as unsigned. Several examples of positive and negative binary two's complement representations are shown in Figure 15.3.

To negate a number in two's complement, invert (complement) all of the bits and add 1 to the result. For example, the 16-bit binary representation of +27 is 00000000 00011011, so we find the representation of -27 by complementing these bits, 11111111 1100100, and then adding 1: 11111111 1100101.

You can tell whether a signed two's complement number is positive or negative by looking at the high-order bit; if it is 1, the number is negative. To find the magnitude of a negative integer, complement the bits and add 1. For example, suppose we are given the binary number 11111111 11010010. It is negative because it starts with a 1 bit. To find the magnitude we complement these bits, 00000000 00101101; add 1 using binary addition, and convert to its decimal form, 00000000 00101110 =  $32 + 8 + 4 + 2 = 46$ . So the original number is -46.

Adding binary numbers is similar to adding decimal values, except that a carry is generated when the sum for a bit position is 2 or greater, rather than 10 or greater, as with decimal numbers. The carry values are represented in binary and may carry over into more than one position as they can in decimal addition.

**Wrap revisited.** Wrap is a representational error that happens when the result of a computation is too large to store in the target variable.

**Issue:** Start with any positive integer. Keep adding 1. Eventually, the answer will be negative because of wrap. Start with any unsigned integer. Keep adding 1. Eventually, the answer will be 0 because of wrap.

With signed integers, wrap happens whenever there is a carry *into* the sign bit (leftmost bit). When  $x$  is the largest positive signed integer we can represent,  $x + 1$  will be the smallest (farthest from 0) negative integer.

The number  $-10$  in binary IEEE format for type `float`:

	sign	exponent	mantissa
$-10.0 = -1.25 \times 2^3 =$	1	1000001 0	1.0100000 00000000 00000000
	31    30    ...    23	22                         ...	0

Figure 15.4. Binary representation of reals.

To be specific, for a 2-byte model, the largest signed value is 32,767, which is represented by the bit sequence  $x = 01111111\ 11111111$ . The value of  $x + 1$  is 10000000 00000000 in binary and  $-32768$  in base 10.

Similarly, with unsigned integers, wrap happens whenever there is a carry *out of* the leftmost bit of the integer. In this case, if  $x$  is the largest unsigned integer,  $x + 1$  will be 0. To be specific, for a 2-byte model, the largest unsigned value is 65,535, which is represented by the bit sequence  $x = 11111111\ 11111111$ . The value, in binary, of  $x + 1$  is 00000000 00000000.

Wrap also can happen when any operation produces a result too large to store in the variable supposed to receive it. Unfortunately, there is no systematic way to detect wrap after it happens. Avoidance is the best policy, and that requires a combination of programmer awareness and caution when working with integers.

#### 15.1.4 Representation of Real Numbers

A real number,  $N$ , is represented by a signed mantissa,  $m$ , multiplied by a base,  $b$ , raised to some signed exponent,  $x$ ; that is,

$$N = \pm m \times b^{\pm x}$$

Inside the computer, each of the components of a real number is stored in some binary format. The IEEE (Institute for Electrical and Electronic Engineers) established a standard for floating-point representation and arithmetic that has been carefully designed to give predictable results with as much precision as possible. Most scientists doing serious numerical computations use systems that implement the IEEE standard. Figure E.3 shows the way that the number  $-10$  is represented according to the IEEE standard for four-byte real numbers. This representation uses 32 bits divided into three fields to represent a real value. The base,  $b = 2$ , is not represented explicitly; it is built into the computer's floating-point hardware.

The mantissa is represented using the sign and magnitude format. The sign of the mantissa (which is also the sign of the number) is encoded in a single bit, bit 31, in a manner similar to that used for integers: A 0 for positive numbers or a 1 for negative values. The magnitude of the mantissa is separated from the sign, as indicated in Figure E.3, and occupies the right end of the number.

After every calculation, the mantissa of the result is *normalized*; that means it is returned to the form  $1.XX\dots X$ , where each  $X$  represents a one or a zero. In this form, the leading bit always is 1 and is always followed by the decimal point. A number always is normalized before it is stored in a memory variable. Since all mantissas follow this rule, the 1 and the decimal point do not need to be stored explicitly; they are built into the hardware instead. Thus the 23 bits in the mantissa of an IEEE real number are used to store the 23  $X$  bits.

In Figure 15.4, the fraction 0.25, or 0.01 in binary, is stored in the mantissa bits and the leading 1 is recreated by the hardware when the value is brought from memory into a register.

When we add or subtract real numbers on paper, the first step is to line up the decimal points of the numbers. A computer using floating-point arithmetic must start with a corresponding operation, denormalization. To add or subtract two numbers with different exponents, the bits in the mantissa of the operand with the smaller exponent must be **denormalized**: The mantissa bits are shifted rightward and the exponent is increased by 1 for each shifted bit position. The shifting process ends when the exponent equals the exponent of the larger operand. We call this number representation *floating point* because the computer hardware automatically "floats" the mantissa to the appropriate position for each addition or subtraction operation.

The precision of a floating-point number is the number of digits that are mathematically correct. Precision directly depends on the number of bits used to store the mantissa and the amount of error that may accumulate due to round-off during computations. Typically a calculation is performed using a few additional bits beyond the lengths of the original operands. The final result then must be rounded off to return it to the length of the

---

$2^{12}$	$2^9$	$2^6$	$2^3$	$2^0$	
↓	↓	↓	↓	↓	
0001	0010	0100	1001		

$$\begin{aligned}
 &= 2^{12} + 2^9 + 2^6 + 2^3 + 2^0 \\
 &= 4096 + 512 + 64 + 8 + 1 = 4681
 \end{aligned}$$
  

$$\begin{aligned}
 0111 &\quad 1011 & 1010 & 0010 &= 2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^9 + 2^8 + 2^7 + 2^5 + 2^1 \\
 &= 16384 + 8192 + 4096 + 2048 + 512 + 256 + 128 + 32 + 2 = 31,650
 \end{aligned}$$


---

**Figure 15.5.** Converting binary numbers to decimal.

operands involved. The different floating-point types use different numbers of bits in the mantissa to achieve different levels of precision. Typical limits are given in Chapter 7.

The exponent is represented in bits 23...30, which are between the sign and the mantissa. Each time the mantissa is shifted right or left, the exponent is adjusted to preserve the value of the number. A shift of one bit position causes the exponent to be increased or decreased by 1. In the IEEE standard real format, the exponent is stored in *excess 127 notation*. Here, the entire 8 bits are treated as a positive value, then the excess value, 127, is subtracted from the 8-bit value to determine the true exponent. In Figure E.3,  $127 + 3 = 130$ , which is the value stored in the eight exponent bits. While this format may be complicated to understand, it has advantages in developing hardware to do quick comparisons and calculations with real numbers.

### 15.1.5 Base Conversion

**Binary to decimal.** We use the table of place values in Figure 15.2 when converting a number from base 2 to base 10. The process is simple and intuitive: Add the place values that correspond to the one bits in the binary representation. The result is the decimal representation (see Figure E.4).

**Binary to and from hexadecimal.** When a programmer must work with numbers in binary or hexadecimal, it is useful to know how to go from one representation to the other. The binary and hexadecimal representations are closely related. Since  $16 = 2^4$ , each hex digit corresponds to 4 bits. Base conversion from hexadecimal to binary (or vice versa) is done by simply expanding (contracting) the number using the table in Figure E.5.

**Decimal to binary.** It also is possible to convert a base-10 number,  $N$ , into a base-2 number,  $T$ , using only the table of place values and a calculator or pencil and paper. First, look at the table in Figure E.1 and find the largest place value that is smaller than  $N$ . Subtract this value from  $N$  and write a 1 in the corresponding position in  $T$ . Keep the remainder for the next step in the process. Moving to the right in  $T$ , write a 1 if the next place value can be subtracted (subtract it and save the remainder) or a 0 otherwise. Continue this process, reducing the remainder of  $N$  until it becomes 0, then fill in all the remaining places of  $T$  with zeros. Figure E.6 illustrates this process of repeated subtraction for both an integer and a real value.

**Hexadecimal to decimal.** Converting a hexadecimal number to a decimal number is analogous to the binary-to-decimal conversion. Each digit of the hexadecimal representation must be converted to its decimal value (for example,  $C$  and  $F$  must be converted to 12 and 15, respectively). Then each digit's decimal value must be multiplied by its place value and the results added. This process is illustrated in Figure E.7.

**Decimal to hexadecimal.** The easiest way to convert a number from base 10 to base 16 is first to convert the number to binary, then convert the result to base 16. The job also can be done by dividing the number repeatedly by 16; the remainder on each division, when converted to a hex digit, becomes the next digit of the answer, going right to left. We do not recommend this method, however, because it is difficult to do in your head and awkward to calculate remainders on most pocket calculators.

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Hexadecimal to binary:  
  
 Use the table to translate each hex digit into a group of four bits.

Binary to hexadecimal: 0011 | 1101 | 1111 | 0101  
  
 Mark off the bits into groups of four, starting at the right.  
 Add leading zeros on the left, if you wish, to make an even group of four bits. Translate each group to one hex digit, using the table.

Hexadecimal to Binary Conversion					
hex	dec	binary	hex	dec	binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

Figure 15.6. Converting between hexadecimal and binary.

$$\begin{array}{r}
 10,542 = 10 \ 1001 \ 1000 \ 1110 \\
 -8,096 = 2^{13} \\
 \hline
 2,446 \\
 -2,048 = 2^{11} \\
 \hline
 398 \\
 -256 = 2^8 \\
 \hline
 142 \\
 -128 = 2^7 \\
 \hline
 14 \\
 -8 = 2^3 \\
 \hline
 6 \\
 -4 = 2^2 \\
 \hline
 2 \\
 -2 = 2^1 \\
 \hline
 0
 \end{array}
 \quad
 \begin{array}{r}
 630.3125 = 10 \ 0111 \ 0110 . 0101 \\
 -512. = 2^9 \\
 \hline
 118.3125 \\
 -64. = 2^6 \\
 \hline
 54.3125 \\
 -32. = 2^5 \\
 \hline
 22.3125 \\
 -16. = 2^4 \\
 \hline
 6.3125 \\
 -4. = 2^2 \\
 \hline
 2.3125 \\
 -2. = 2^1 \\
 \hline
 0.3125 \\
 -0.25 = 2^{-2} \\
 \hline
 0.0625 \\
 -0.0625 = 2^{-4} \\
 \hline
 0
 \end{array}$$

Figure 15.7. Converting decimal numbers to binary.

$$\begin{aligned}
 2A3C &= 2*16^3 + 10*16^2 + 3*16 + 12 \\
 &= 2*4096 + 10*256 + 48 + 12 = 10,812 \\
 BFD &= 11*16^2 + 15*16 + 13 \\
 &= 11*256 + 240 + 13 = 2,816 + 253 = 3,069 \\
 A2.D2 &= 10*16^1 + 2*16^0 + 13*16^{-1} + 2*16^{-2} \\
 &= 160 + 2 + .8125 + 0.0078125 = 162.8203125
 \end{aligned}$$

Figure 15.8. Converting hexadecimal numbers to decimal.

Character literals can be written symbolically as an escape character followed by the letter `x` and the character's hexadecimal code from the ASCII table:

Meaning	Symbol (portable)	Hex Escape Code (ASCII only)
The letter A	'A'	'\x41'
The letter a	'a'	'\x61'
Newline	'\n'	'\xA'
Formfeed	'\f'	'\xC'
Blank space	' '	'\x20'
Escape character	'\\'	'\x1B'
Null character	'\0'	'\x00'

Figure 15.9. Hexadecimal character literals.

## 15.2 Hexadecimal Input and Output

Input and output in binary notation are not directly supported in C++ because binary numbers are difficult for human beings to read and use accurately. We do not work well with rows of bits; they make our eyes swim. Instead, all input and output is done in decimal, octal, or hexadecimal notation. When we use `>>` to read input into an integer variable, the type of the variable tells the system to read an integer, and the state of the stream flags tells whether that will be converted to binary using base 10 or base 16. All numbers are stored in binary. The default stream setting is to use base 10, and the stream manipulators `hex`, `dec` and `oct` (for octal) are used to change the status flags.

The opposite conversion (binary to a string of digits) is done by `<<` when we output a number. These conversions are done automatically, and we often forget they happen. But on some occasions, base-10 representation is not convenient for input or is confusing for output. For example, if a programmer needs to write a mask value or a disk address, the value should be written as an unsigned integer and it is often easiest to work with hexadecimal form.

**Reading and writing integers in hexadecimal notation.** The program in Figure ?? demonstrates how to read and print unsigned integers using both decimal and hex notations. By showing numbers in both base 10 and base 16, we hope to build some understanding of the numbers and their representations.

### Notes on Figure 15.10: I/O for hex and decimal integers.

#### *First box: Input in base 10.*

- The input for an unsigned integer is expected to be a sequence of decimal digits with no leading sign.
- The syntax is the same for reading a `short int`, an `int`, and a `long int`.
- The input variable's type determines what happens; if the number that is read is too large to store in the variable, the maximum value for the variable's type is stored instead.
- Whether the input was in decimal or hex notation, and whether the hex digits were entered in upper or lower case, the output will be printed according to whether the stream is in `hex` or `dec` or `oct` mode.
- Warning: as is shown here, if you use `hex` or `oct`, *always put the stream back into dec mode*.
- Note that a number bigger than 7 looks larger printed in octal than in decimal, and larger in decimal than in hex.
- A sample output from this box is:

```
Please enter an unsigned int: 345
= 345 in decimal and = 159 in hex, and 531 in octal.
```

- Here is an input that is too large to store in the variable. When this happens, the stream's `fail` flag is turned on and the variable is set to the maximum value for its type. Note the `ffff` printed in hex – that is the maximum number that can be stored in a `short int`.

```
Please enter an unsigned int: 66000
= 65535 in decimal and = ffff in hex, and 177777 in octal.
```

- When a negative number is entered, instead of an unsigned number, the stream's `fail` flag is turned on and the variable is set to 0, as in the following output:

---

This program demonstrates how unsigned integers of various sizes may be read and written.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main( void )
{
    unsigned short ui, xi;

    cout <<"\n Please enter an unsigned int:  ";
    cin >>dec >>ui;
    cout <<" = " <<ui <<" in decimal and = "
        <<hex <<ui <<" in hex, and = "
        <<oct <<ui <<" in octal.\n" <<dec;

    cout <<"\n Please enter an unsigned int in hex:  ";
    cin >>hex >>xi ;
    cout <<" = " <<xi <<" in decimal and = "
        <<hex <<xi <<" in hex, and = "
        <<oct <<xi <<" in octal.\n\n" <<dec;

    return 0;
}
```

---

Figure 15.10. I/O for hex and decimal integers.

```
Please enter an unsigned int: -31
= 0 in decimal and = 0 in hex.
```

*Second box: Input in hexadecimal.*

- To cause input to be in hexadecimal, set the input stream to `>>hex`.
- Hexadecimal input may include any mixture of digits from 1 to 9 and letters from a to f or from A to F.

```
Please enter an unsigned int in hex: aBc
= 2748 in decimal and = abc in hex, and = 5274 in octal.
```

- The input may have a leading 0x to indicate hex, but it is also acceptable to omit the 0x:

```
Please enter an unsigned int in hex: 0x12AB
= 4779 in decimal and = 12ab in hex, and = 11253 in octal.
```

### 15.3 Bitwise Operators

C and C++ include six operators whose purpose is to manipulate the bits inside a byte. For most applications, we do not need these operators. However, systems programmers use them extensively when implementing type conversions and hardware interfaces. Any program that must deal directly with a hardware device may need to pack bits into the instruction format for that device or unpack status words returned by the device. Another application is an archive program that compresses the data in a file so that it takes up less space as an archive file. A program that works with the DES encryption algorithm makes extensive use of bitwise operators to encode or decode information.

Bitwise operators fall into two categories: shift operators move the bits toward the left or the right within a byte, and **bitwise-logic** operators can turn individual bits or groups of bits on or off or **toggle** them. These operators are listed in Figure 15.11. The operands of all these operators must be one of the integer types (type `unsigned` or `int` and length `long`, `short`, or `char`). Because we cannot use the subscript operator to reference bits individually, we must use some combination of the bitwise operators to extract or change the value of a given bit in a byte.

Arity	Symbol	Meaning	Precedence	Use and Result
Unary	$\sim$	Bitwise complement	15	Reverse all bits
Binary	$<<$	Left shift	11	Move bits left
	$>>$	Right shift	11	Move bits right
	$\&$	Bitwise AND	8	Turn bits off or decompose
	$\wedge$	Bitwise exclusive OR (XOR)	7	Toggle bits
	$ $	Bitwise OR	6	Turn bits on or recombine

Figure 15.11. Bitwise operators.

### 15.3.1 Masks and Masking

When a program manipulates codes or uses hardware bit switches, it often must isolate one or more bits from the other bits that are stored in the same byte. The process used to do this job is called *masking*. In a masking operation, we use a bit operator and a constant called a **mask** that contains a bit pattern with a 1 bit corresponding to each position of a bit that must be isolated and a 0 bit in every other position.

For example, suppose we want to encode a file so that our competitors cannot understand it (a process called *encryption*). As part of this process, we want to split a short integer into four portions, A (3 bits), B (5 bits), C (4 bits), and D (4 bits), and put them back together in the scrambled order: C, A, D, B. We start by writing out the bit patterns, in binary and hex, for isolating the four portions. The hex version then goes into a `#define` statement. The table in Figure 15.12 shows these patterns. Then we use the  $\&$  operator with each mask to decompose the data, shift instructions to move the pieces around, and  $|$  operations to put them back together in the new order. This simple technique for **encoding and decoding** is illustrated in the next section in Figures 15.22 and 15.23.

**Bitwise AND ( $\&$ ) turns bits off.** The bitwise AND operator,  $\&$ , is defined by the fourth column of the truth table shown in Figure 15.13, which is the same as the truth table for logical AND. The difference is that logical operators are applied to the truth values of their operands, and bitwise operators are applied to each pair of corresponding bits in the operands. Figure 15.14 shows three examples of the use of bitwise AND. To get the answer (on the bottom line) look at each pair of corresponding bits from the two operands above it, and apply the bitwise AND truth table to them.

The bitwise AND operator can be used with a mask to isolate a bit field by “turning off” all the other bits in the result. (None of the bitwise operators affect the value of the operand in memory.) The first column in Figure 15.14 shows that using a mask with 0 bits will turn off the bits corresponding to those zeros. The second column shows how a mask with a field of four 1 bits can be used to isolate the corresponding field of  $x$ . The third column of this figure illustrates the semantics of  $\&$  by pairing up all combinations of 1 and 0 bits.

Part	Bit Pattern	<code>#define</code> , with Hex Constant
A	11100000 00000000	<code>#define A 0xE000</code>
B	00011111 00000000	<code>#define B 0x1F00</code>
C	00000000 11110000	<code>#define C 0x00F0</code>
D	00000000 00001111	<code>#define D 0x000F</code>

Figure 15.12. Bit masks for encrypting a number.

x	y	$\sim x$	$x \& y$	$x   y$	$x \wedge y$
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Figure 15.13. Truth tables for bitwise operators.

---

The bit vectors used here are 1 byte long. A result bit is 1 if both operand bits are 1.

Binary	Hex	Binary	Hex	Binary	Hex
x	1111 1111	0xFF	x	0111 1010	0x7A
mask	0010 0000	0x20	mask	1111 0000	0xF0
x & mask	0010 0000	0x20	x & mask	0111 0000	0x70

Figure 15.14. Bitwise AND ( $\&$ ).

Note that a bit in the result is 1 only if both of the corresponding bits in the operands are 1.

**Bitwise OR (|) turns bits on.** Just as the bitwise AND operator can be used to turn bits off or decompose a bit vector, the bitwise OR operator, defined in the fifth column of Figure 15.13, can be used to turn on bits or reassemble the parts of a bit vector. Figure 15.15 shows three examples of the use of this operator. The first column shows how a mask can be used with | to turn on a single bit. The second column shows how two fields in different bit vectors can be combined into one vector with bitwise OR. The third column of this figure illustrates the semantics of | by pairing up all combinations of 1 and 0 bits. Note that a bit in the result is 1 if either of the corresponding bits is 1.

**Complement ( $\sim$ ) and bitwise XOR ( $\wedge$ ) toggle bits.** Sometimes we want to turn a bit on, sometimes off, and sometimes we want to simply change it, whatever its current state is. For example, if we are implementing a keyboard handler, we need to change from upper-case to lower-case or vice versa if the user presses the Caps Lock key. When we change a switch setting like this, we say we *toggle* the switch. The complement operator,  $\sim$ , defined by the third column in Figure 15.13, toggles all the bits in a word, turning 1's to 0's and 0's to 1's.

The bitwise XOR operator ( $\sim$ ) can be used with a mask to toggle some of the bits in a vector and leave the rest unchanged. This is essential when several switches are packed into the same control word. The first two columns of Figure 15.16 show how to use XOR with a mask to toggle some of the bits in a vector but leave others unaffected. Of course, to actually change the switch setting, you must use = to store the result back into the switch variable. The last column of this figure illustrates the semantics of  $\sim$  by pairing up all combinations of 1 and 0 bits. Note that a bit in the result is 1 if the corresponding bits in the operands are *different*.

**The three negations.** C has three unary operators that are alike but different. Figure 15.17 shows the operation of these operators: logical NOT, complement, and arithmetic negation. All of them compute some kind of opposite, but on most machines, these three “opposites” are different values. Normally, this is no problem, since you use the three operators for different types of operands and in different situations. This discussion is included for those who are curious about why we need three ways to say no.

---

The bit vectors used here are 1 byte long. A result bit is 1 if either operand bit is 1.

Binary	Hex	Binary	Hex	Binary	Hex
x	0000 0000	0x00	x	0000 1100	0x0C
mask	0010 0000	0x20	mask	1111 0000	0xF0
x   mask	0010 0000	0x20	x   mask	1111 1100	0xFC

Figure 15.15. Bitwise OR (|).

---

The bit vectors used here are 1 byte long. A result bit is 1 if the operand bits are different.

Binary	Hex	Binary	Hex	Binary	Hex
x	1111 1111	0xFF	x	1100 1100	0xCC
mask	0011 0000	0x30	mask	1111 0000	0xF0
x $\wedge$ mask	1100 1111	0xCF	x $\wedge$ mask	0011 1100	0x3C

Figure 15.16. Bitwise XOR ( $\wedge$ ).

We illustrate the results of the three ways to say no: logical NOT, complement, and negate. The bit vectors used here are 1 byte long, the representation for negative numbers is two's complement.

Decimal	Hex	Binary	Decimal	Hex	Binary	Decimal	Hex	Binary
x 0	0x00	0000 0000	x 1	0x01	0000 0001	x -10	0xF6	1111 0110
!x 1	0x01	0000 0001	!x 0	0x00	0000 0000	!x 0	0x00	0000 0000
~x -1	0xFF	1111 1111	~x -2	0xFE	1111 1110	~x 9	0x09	0000 1001
-x 0	0x00	0000 0000	-x -1	0xFF	1111 1111	-x 10	0xA	0000 1010

Figure 15.17. Just say no.

The logical NOT operator (!) is the simplest (see the second row in Figure 15.17). It turns true values to `false` and false values to `true`. True is represented by the *integer 1*, which has many 0 bits and only a single 1 bit.

The complement operator, which toggles all the bits, is shown on the third row in Figure 15.17. Looking at the first column of Figure 15.17, note how the result of `!x` is very different from the result of `~x`.

Most modern computers use two's complement notation to represent negative integers. The **two's complement** of a value is always 1 greater than the bitwise complement (also known as one's complement) of that number.<sup>3</sup> We can see this relationship by comparing the third and fourth lines in Figure 15.17.

### 15.3.2 Shift Operators

Once we have used & to decompose a bit vector or isolate a switch setting, we generally need to move the bits from the middle of the result to the end before further processing. Conversely, to set a switch or recompose a word, bits usually need to be shifted from the right end of a word to the appropriate position before using | to recombine them. This is the purpose of the right-shift (>>) and left-shift (<<) operators.

The left operand of a shift operator can be either a signed or unsigned integer. This is the bit vector that is shifted. The right operand must be a nonnegative integer;<sup>4</sup> it is the number of times that the vector will be shifted by one bit position. Figure 15.18 shows examples of both shift operations.

**Left shifts.** Left shifts move the bits of the vector to the left; bits that move off the left end are forgotten, and 0 bits are pulled in to fill the right end. This is a straightforward operation and very fast for hardware. Shifting left by one position has the same effect on a number as multiplying by 2 but it is a lot faster for the machine. In the table, you can see that the result of `n << 2` is four times the value of `n`, as long as no significant bits fall off the left end.

<sup>3</sup>This is the definition of *two's complement*. Appendix E explains this topic in more detail.

<sup>4</sup>The C standard specifies that negative shift amounts are undefined.

The table uses 1-byte variables declared thus:

```
signed char s; // A one-byte signed integer.
unsigned char u; // A one-byte unsigned integer.
```

Signed	Decimal	Hex	Binary	Unsigned	Decimal	Hex	Binary
s 15	0x0F	0000 1111	u 10	0xA	0000 1010		
s << 2	60	0x3C	0011 1100	u << 2	40	0x28	0010 1000
s >> 2	3	0x03	0000 0011	u >> 2	2	0x02	0000 0010
s -10	0xF6	1111 0110	u 255	0xFF	1111 1111		
s << 2	-40	0xD8	1101 1000	u << 2	252	0xFC	1111 1100
s >> 2	-3	0xFD	1111 1101	u >> 2	63	0x3F	0011 1111

Figure 15.18. Left and right shifts.

---

**Problem scope:** Read a 32-bit Internet IP address in the form used to store addresses internally and print it in the four-part dotted form that we customarily see.

**Input:** A 32-bit integer in hex. For example: `fa1254b9`

**Output required:** The dotted form of the address. For the example given, this is: `250.18.84.185`

---

**Figure 15.19. Problem specifications: Decoding an Internet address.**

One thing to remember is that a shift operation does not change the number of bits in a vector. If you start with a 16-bit vector and shift it 10 places to the left, the result is a 16-bit vector that has lost its 10 high-order bits and has had 10 0 bits inserted on the right end.

**Right shifts.** Unfortunately, right shifts are not as simple as left shifts. This is because computer hardware typically supports two kinds of right shifts, signed right shifts and unsigned right shifts. In C, an **unsigned shift** always is used if the operand is declared to be unsigned. A **signed shift** is used for signed operands.<sup>5</sup>

A signed right shift fills the left end with copies of the sign bit (the leftmost bit) as the number is shifted, and an unsigned shift fills with 0 bits. The reason for having signed shifts at all is so that the operation can be used in arithmetic processing; with a signed shift, the sign of a number will not be changed from negative to positive when it is shifted. There is no difference between the effects of signed and unsigned shifts for positive numbers, since all have a 0 bit in the leftmost position anyway.

Analogous to left shift, a right shift by one position divides a number by 2 (and discards the remainder). A right shift by  $n$  positions divides a number by  $2^n$ . In the table, you can see that `15 >> 2` gives the same result as `15/4`.

### 15.3.3 Example: Shifting and Masking an Internet Address

IP (Internet protocol) addresses normally are written as four small integers separated by dots, as in `32.55.1.102`. The machine representation of these addresses is an unsigned long integer. Figure 15.19 lists the specifications for decoding an IP address. The program in Figure 15.20 takes the hex representation of an address, decodes it, and prints it in the four-part dotted form we are accustomed to reading.

#### Notes on Figure 15.20: Decoding an Internet address.

**First box: the mask.** We define a mask to isolate the last 8 bits of a long integer. We can write the constant with or without leading zeros, as `0xffL` or `0x000000ffL`, but we need the `L` to make it a long integer.

#### Second box: the variables.

- We declare the input variable as `unsigned` so that unsigned shift operations will be used and `long` because the input has 32 bits.
- We use each of the other four variables to hold one field of the IP address as we decompose it. These could be short integers since the values are in the range `0...255`.

#### Third box: hex input and output.

- We put the input stream into hex mode before reading the input. When done, we put it back into decimal mode.
- To format the output so that all 8 hex digits are displayed, it is necessary to set a field width of 8 columns and to set the fill character to '`'0'`'. The fill character will stay set until explicitly changed. The field width must be set separately for each thing printed.

#### Fourth box: byte decomposition.

- The coded address contains four fields; each is 8 bits. The byte mask lets us isolate the rightmost 8 bits in an address. In this box, we store each successive 8-bit field of the input into one of the `f` variables.
- To get `f1`, the first field of the IP address, we shift the address 24 places to the right so that all but the first 8 bits are discarded.

---

<sup>5</sup>According to the standard, an implementor could choose to use an unsigned shift in both cases. However, signed shifts normally are used for signed values.

---

The problem specifications are in Figure 15.19.

```
#include <iostream>
using namespace std;

#define BYTEMASK 0xffL           The L is to make a long integer.

int main( void )
{
    unsigned long ipAddress;
    unsigned f1, f2, f3, f4;

    cout <<"\n Please enter an IP address as 8 hex digits: ";
    cin >>hex >>ipAddress >>dec;
    cout <<"\t You have entered " <<setw(8) <<setfill('0')
        <<hex <<ipAddress <<endl;

    f1 = ipAddress >> 24 & BYTEMASK;
    f2 = ipAddress >> 16 & BYTEMASK;
    f3 = ipAddress >> 8 & BYTEMASK;
    f4 = ipAddress           & BYTEMASK;

    cout <<dec <<"\t The IP address in standard form is: "
        <<f1 <<"." <<f2 <<"." <<f3 <<"." <<f4 <<endl;
}
```

---

**Figure 15.20.** Decoding an Internet address.

- If all machines used 32-bit long integers, we would not need the masking operation to compute `f1`. We include it to be sure that this code can be used on machines with longer integers.
- To get `f2` and `f3`, we shift the input by smaller amounts and mask out everything except the 8 bits on the right.
- Since `f4` is supposed to be the rightmost 8 bits of the IP address, we can mask the input directly, without shifting.

The output from two sample runs is

```
Please enter an IP address as 8 hex digits: fa1254b9
You have entered fa1254b9
The IP address in standard form is: 250.18.84.185
-----
Please enter an IP address as 8 hex digits: 0023fa01
You have entered 0023fa01
The IP address in standard form is: 0.35.250.1
```

## 15.4 Application: Simple Encryption and Decryption

We have enough tools to write a simple application program that uses the bitwise operators. Let us return to the encryption example<sup>6</sup>. Using the masks in Figure 15.12, we encrypt a short integer by breaking it into four portions, A (3 bits), B (5 bits), C (4 bits), and D (4 bits), and putting them back together in a scrambled order: C, A, D, B. That is, we start with the bits in this order: `aaabbbaaaaaaaa` and end with the scrambled order: `cccccaaaaddddbbbb`. Figure 15.21 specifies the problem, Figure 15.22 is the encryption program, and

---

<sup>6</sup>Please note that this is just an programming example; it is not good cryptography. Nonprofessionals should never try to invent cryptosystems – the result will not be secure.

**Problem scope:** Write a function to encrypt a short integer and a main program to test it. Write a matching decryption program.

**Input:** Any short integer.

**Output required:** The input will be echoed in both decimal and hex; the encrypted number also will be given in both bases.

**Formula:** Divide the 16 bits into four unequal-length fields (3, 5, 4, and 4 bits) and rearrange those fields within the space as shown:

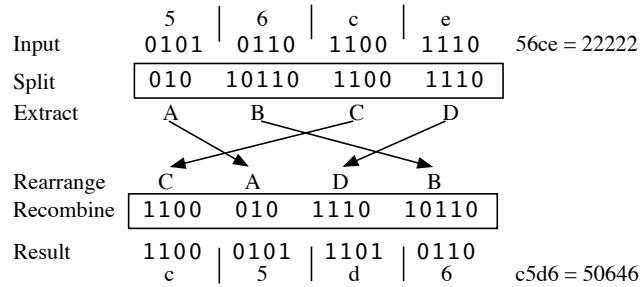


Figure 15.21. Problem specifications: Encrypting and decrypting a number.

Figure 15.23 is the program for subsequent decryption. The encrypted results of this program might fool your friends, but it is not a secure system.

### Notes on Figures 15.22 and 15.23. Encrypting and decrypting a number.

#### *First boxes: the masks.*

- We refer to the bit positions with numbers 0...15, where bit 15 is the leftmost and bit 0 is the last bit on the right.
- The diagram in Figure 15.21 shows the positions of the four fields and the names that are used in the program for those positions.
- We define one bit mask for each part of the number we want to isolate. The hex constants for encryption are developed in Figure 15.12.
- The bit masks for the decryption process allow us to isolate the same four fields in different positions within the word so that we can reverse the encryption process.

#### *Second boxes: the interfaces for functions encrypt() and decrypt().*

- We use type `unsigned` for all bit manipulation, so that the leftmost bit is treated just like any other bit. (With signed types, the leftmost bit is treated differently because it is the sign.)
- We are working with 16-bit numbers, so we use type `short`. Therefore, the argument type and the return type are both `unsigned short`.
- When a signed short integer is cast to an unsigned short integer, or vice versa, no bits change within the word. This is guaranteed by the language standard for the cast from signed to unsigned. It is officially “undefined” for casting from unsigned to signed. However, on a 2’s complement machine, the result will almost certainly be no change in the bit pattern. Thus, if we cast a signed int to an unsigned, then back to signed int, the result will be the same number we started with.

#### *Third boxes: unpacking.*

- The `&` operation turns off all the bits of `n` except those that correspond to the 1 bits in the mask. This lets us isolate one bitfield so we can store it in a separate variable. For example, here we start with 22222 = 56ce and apply the b-mask to get field b of the word:

&	22222 = 56ce:	0101	0110	1100	1110	
	BE = 0x1f:	0001	1111	0000	0000	
	field b:	0001	0110	0000	0000	

- We use four `&` expressions, one for each segment of the data. We call this process **unpacking** the fields.
- After masking, we shift the isolated bits to their new positions. To compute the shift amount, we subtract the original position of the first bit in the field from its new position. We use `>>` for positive shift amounts, `<<` for negative amounts.

before >> 8	0001	0110	0000	0000	
after >> 8	0000	0000	0001	0110	

This program carries out the first half of the specifications in Figure 15.21.

```
#include <iostream>
using namespace std;

#define AE 0xE000      // bits 15:13 end up as 11:9
#define BE 0x1F00      // bits 12:8 end up as 4:0
#define CE 0x00F0      // bits 7:4 end up as 15:12
#define DE 0x000F      // bits 3:0 end up as 8:5

unsigned short encrypt( unsigned short n );

int main( void )
{
    short in;
    unsigned short crypt;

    cout <<"Enter a short number to encrypt:  ";
    cin >>in;

    // Cast the int to unsigned before calling encrypt.
    crypt = encrypt( (unsigned short) in );

    cout <<"\nThe input number in base 10 is:  " <<in <<" \n"
        <<"The input number in hexadecimal is:  " <<hex <<in <<" \n\n"
        <<"The encrypted number in base 10 is:  " <<dec <<crypt <<"\n"
        <<"The encrypted number in base 16 is:  " <<hex <<crypt <<"\n\n";
}

// -----
unsigned short
encrypt( unsigned short n )
{
    unsigned short a, b, c, d;                      // for the four parts of n

    a = (n & AE) >> 4;    // Isolate bits 15:13, shift to positions 11:9.
    b = (n & BE) >> 8;    // Isolate bits 12:8, shift to positions 4:0.
    c = (n & CE) << 8;    // Isolate bits 7:4, shift to positions 15:12.
    d = (n & DE) << 5;    // Isolate bits 3:0, shift to positions 8:5.

    return c | a | d | b ;           // Pack the four parts back together.
}
```

Figure 15.22. Encrypting a number.

- We use four mask-and-shift expressions, one for each segment of the data.

**Fourth boxes: packing.**

- We use the `|` operator to put the four parts back together. We call this process *packing*.

a: 0000 0100 0000 0000	b: 0000 0000 0001 0110
c: 1100 0000 0000 0000	d: 0000 0001 1100 0000
<hr/>	
Result: 1100 0101 1101 0110    c5d6 = 50646	

- It does not matter in what order the operands are listed in the `|` expression, we get the same result whether we write `a | b | c | d` or `c | a | d | b`.

Decryption is the inverse of encryption. This program looks very much the same as the encryption program in Figure 15.22 except that every operation is reversed.

```
#include <stdio.h>

#define AD 0x0E00          // bits 11:9 goto 15:13
#define BD 0x001F          // bits 4:0 goto 12:8
#define CD 0xF000          // bits 15:12 goto 7:4
#define DD 0x01E0          // bits 8:5 goto 3:0

unsigned short decrypt( unsigned short n );

int main( void )
{
    unsigned short in;
    short decrypted;;
    cout <<"Enter an encrypted short int in hex:  ";
    cin >>hex >>in;
    decrypted = (signed short)decrypt( in );

    cout <<"\nThe input number in base 10 is:  " <<in <<"\n"
    <<"The input number in hexadecimal is:  " <<hex <<in <<" \n\n"
    <<"The decrypted number in base 10 is:  " <<dec <<decrypted <<" \n"
    <<"The decrypted number in base 16 is:  " <<hex <<decrypted <<" \n\n";
    return 0; }

// -----
unsigned short                      // use unsigned for bit manipulation
decrypt( unsigned short n )
{
    unsigned short a, b, c, d;           // for the four parts of n

    a = (n & AD) << 4; // Isolate bits 4:6, shift to positions 0:2.
    b = (n & BD) << 8; // Isolate bits 11:15, shift to positions 3:7.
    c = (n & CD) >> 8; // Isolate bits 0:3, shift to positions 8:11.
    d = (n & DD) >> 5; // Isolate bits 7:10, shift to positions 12:15.

    return a | b | c | d ;           // Pack the four parts back together.
}
```

Figure 15.23. Decrypting a number.

*Encryption.*

- Here we encrypt the number shown in the diagrams:

```
Enter a short number to encrypt: 22222
```

```
The input number in base 10 is: 22222
The bits of the input number are: 56ce
```

```
The encrypted number in base 10 is: 50646
The encrypted number in base 16 is: c5d6
```

- This encryption process shuffles the bits in a word without changing any of them. That is what makes it easy to reverse and decrypt. If all the bits are the same, the encrypted value will be exactly like the original value. So if the input number is -1, its hex representation is **ffff**, and the encrypted value is also **ffff**. Similarly 0 (or 0000) gets transformed to 0000.

- Here is a negative number:

```
Enter a short number to encrypt: -22222
```

```
The input number in base 10 is: -22222
The bits of the input number are: a932
```

```
The encrypted number in base 10 is: 14921
The encrypted number in base 16 is: 3a49
```

*Decryption.*

- The output from decryption of the second example is

```
Enter an encrypted short int in hex: 3a49
```

```
The input number in base 10 is: 14921
The input number in hexadecimal is: 3a49
```

```
The decrypted number in base 10 is: -22222
The decrypted number in base 16 is: a932
```

## 15.5 Bitfield Types

A bitfield declaration defines a structured type in which the fields are mapped onto groups of bits rather than groups of bytes. **Bitfield structure** types are useful in programs that interface with and manipulate hardware devices, turning on and turning off individual bits in specific positions at fixed memory addresses. The bitfield declaration lets us represent and manipulate such devices symbolically, an important aid to processing them correctly.

A bitfield declaration has the same elements as any **struct** declaration, with the restriction that the base type of each field is an unsigned integer (char, short, int, or enum) and there is a colon and an integer (field width, in bits) after each field name. The field name is optional. Fields can be any width, and the total size of a bitfield structure may exceed one byte. The run-time system packs and unpacks the fields for you, enabling the programmer to use the names of the fields without concern for where or how they are stored.

The precise rules and restrictions for bitfield declarations are implementation dependent, as is the order in which the fields will be laid out in memory. This is unavoidable: computer architecture is not at all standardized and dealing with memory below the byte level obviously involves the nonstandard aspects of the hardware.

A big endian computer stores the high-order byte of an **int** at the lowest memory address and stores the bytes in order with the low-order end of the number at the highest memory address. A little endian machine stores bytes in memory in exactly the opposite order! In both kinds of machines, numbers in the CPU's registers are stored in big-endian order. Thus, anything that relies on byte order in a computer is only portable to other machines of the same basic architecture. The current Intel machines are little-endian.

The program in Figure 15.24 can be used to determine how your own hardware and compiler treat bitfields. It declares a bitfield object, initializes it, prints various information about it, and returns a hexadecimal dump

of the contents so that the placement of information within the object can be determined. The packing diagram at the bottom of the figure shows the way this structure is implemented by one compiler (gcc) for one big-endian machine architecture. Its treatment of bitfields is simple and logical. The results of this program on a modern little-endian machine are very different and very difficult to understand.

**Notes on Figure 15.24. Bitfield-structure demonstration.**

**First box: a bitfield type.**

- We declare a bitfield type, `DemoType`, that occupies a minimum of 37 bits. The compiler will add padding, as necessary, to meet the requirements of the local hardware.
- Here are two diagrams of how the data might be laid out in a bitfield structure. Byte boundaries are indicated by divisions in the white strip at the bottom. Short-word boundaries are dark lines that cross the gray area, and field boundaries are indicated by the white boxes in the gray area. This implementation

---

This program can be used to determine how a compiler and its underlying hardware treat bitfields. The packing used by the Gnu C compiler is diagrammed below.

```
#include <iostream>

struct DemoType {
    unsigned int one    : 1;
    unsigned int two    : 3;
    unsigned int three  :10;
    unsigned int four   : 5;
    unsigned int       : 2;
    unsigned int five   : 8;
    unsigned int six    : 8;
};

int main( void )
{
    int k;
    unsigned char* bptr;
    DemoType bit = { 1, 5, 513, 17, 129, 0x81 };

    printf( "\n sizeof DemoType = %lu\n", sizeof(DemoType) );

    printf( "initial values: bit = %u, %u, %u, %u, %u, %u\n",
            bit.one, bit.two, bit.three, bit.four, bit.five, bit.six );
    bptr = (unsigned char*)&bit;
    printf( " hex dump of bit: %02x %02x %02x %02x %02x %02x %02x\n",
            bptr[7], bptr[6], bptr[5], bptr[4], bptr[3], bptr[2], bptr[1], bptr[0] );

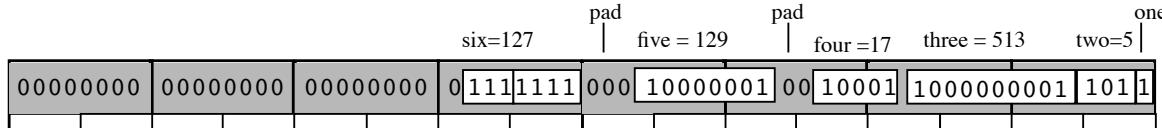
    bit.three = 1023;
    printf( "\n assign 1023 to bit.three: %u, %u, %u, %u, %u, %u\n",
            bit.one, bit.two, bit.three, bit.four, bit.five, bit.six );
    k = bit.two;
    printf( "assign bit.two to k: k = %i\n", k );

    return 0;
}
```

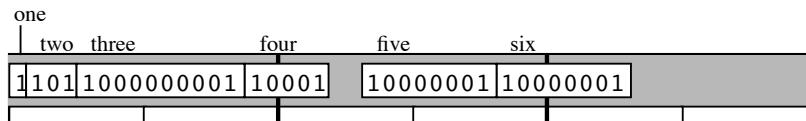
---

**Figure 15.24. Bitfield-structure demonstration.**

packs the bits tightly together starting at the left. Fields three and five cross byte boundaries, field four crosses a short-word boundary, and field six crosses a long-word boundary.



- The diagram above shows the way the fields are laid out on a current little-endian machine.
- The diagram below shows the way the fields were laid out on an older big-endian machine. The two layouts are very different! In both cases, the bytes are shown as they would appear in a machine register.



- The standard permits the fields to be assigned memory locations in left-to-right or right-to-left order and padding to be added wherever necessary to meet alignment requirements on the local system.
- Two bits of padding are added between fields four and five because of the unnamed 2-bit field declared there. This is the only instance in the C language where a memory location can be declared without associating a name with it.
- Additional padding bits were placed after field six to fill the structure up to a word boundary, as required by the local compiler.
- One, but not both compilers added padding between fields five and six to avoid having field six cross a long-word boundary.
- The third box prints out the size of the structure on the local system. In the older implementation, it was 6 bytes (48 bits). Therefore, the last 11 bits are padding. Since this system uses *short-word alignment*, all objects occupy an even number of bytes and begin on an even-numbered memory address.
- My current little-endian computer, uses long-word alignment, and the size of the same type is 8 bytes.

#### *Second box: a bitfield object.*

- We declare a `DemoType` object named `bit`, and initialize its fields to distinctive values. The syntax for initialization is like the syntax for an ordinary struct, except that no initializer is provided for an unnamed field.
- Each initializer value has the correct number of bits needed to fill its field. All but one of these values starts and ends with a 1 bit and has 0 bits in between. When we look at this in hexadecimal, we can see where each field starts and ends. From this information, we can determine the packing and padding rules used by the local compiler.
- A 2-bit unnamed field lies between fields four and five. An initializer should not supply a value for a padding field; the compiler may store anything there that is convenient. Our compiler initializes these fields to 0 bits.

#### *Third box: The size of the object .*

- Please note that C formatted output is being used here. The entire C language is a subset of C++.
- The reason for using C output is simplicity. It is simply easier and briefer to deal with low-level concerns using a low-level language. Here, we want the bytes of the object printed with two columns per byte, separated by spaces. This is easy using `%02x`, and very difficult using `<<`.

#### *Fourth box: making the packing order visible.*

In this box, we use a pointer cast and subscripts to look at the contents of each byte of the bitfield object.

- First we print out the contents of each field, simply to verify that the initializer works as expected. This is not always the case. Initializer values too large to fit into the corresponding fields are simply truncated (the leftmost bits are dropped). This is the output from the program in Figure 15.24 when compiled under the `clang` compiler for a Mac running OS 10.11.

```

sizeof DemoType = 8
initial values: bit = 1, 5, 513, 17, 129, 127
hex dump of bit: 00 00 00 7f 10 24 60 1b

```

- We next use a pointer, to enable us to look at the individual bytes of the bitfield object. At the top of `main()`, we declare `bptr` to be `unsigned` because we want to print the values in hexadecimal. We further declare it to be of type `char*` so that we can print individual bytes of the bit vector.
- By using a pointer cast, and storing the result in `bptr`, we are able to use `bptr` with a subscript to access each byte of the bitfield object.
- We cast the address of `bit` to the type of `bptr` and set `bptr` to point at the beginning of `bit`. Then we print out the bytes to verify the position of each field in the whole.
- The 2 in the conversion specifier (`%02x`) causes the values printed to be two columns wide, while the 0 before it means that leading zeros will be printed. This is the form in which the hexadecimal codes are easiest to convert to binary.
- From the first line of output, we know that `bit` occupies 8 bytes in the current implementation. So we use `bptr` to print 8 bytes starting at the address of the beginning of `bit`. Any value of any type can be printed in hex in this way, by using an `unsigned char*`.
- The array is printed backwards (from byte 7 to byte 0) because this code was written and run on a little-endian machine. On a big-endian machine, the subscripts would start at 0 and increase.
- Below we rewrite the hex value from each output field in binary, with spaces separating the bits into groups of four that correspond to the hexadecimal output. We omit the last byte of zeros. The conversion values simply are filled in using the table values in Appendix E, Figure E.4. A `v` is written above the first bit of each of the 6 bytes. Long-word boundaries are marked by `L`, and short-word boundaries by `S`:

00	00	00	7f	10	24	60	1b							
L		S		L		S		L						
v	v	v	v	v	v	v	v	v						
0000	0000	0000	0000	0000	0111	1111	0001	0000	0010	0100	1100	0000	1000	1011

We now rewrite the bits using spaces to separate the bitfields. The decimal value of each bitfield is written below it. Padding bits are marked by `o`. We produced the diagram on the prior page from this output.

```

oooooooo oooooooo oooooooo o 1111111 ooo 10000001 oo 1001 1000000001 101 1
                    127           129           17           513           5   1

```

**Fifth box: using an individual field.** In this box, we use the symbolic part names to look at the logical values in the structure.

- The assignment to `bit.three` demonstrates that an integer value may be assigned to a bitfield in the ordinary way. The C run-time system positions the bits correctly and assigns them to the structure without disturbing the contents of other fields.
- The assignment `k = bit.two` shows that a bitfield may be assigned to an integer in the ordinary way. The C run-time system will lengthen the bitfield to the size of an integer.

```

assign 1023 to bit.three: 1, 5, 1023, 17, 129, 129
assign bit.two to k: k = 5

```

### 15.5.1 Bitfield Application: A Device Controller (Advanced Topic)

An artist has a studio with a high sloping ceiling containing skylights. Outside, each skylight is covered with louvers that can be opened fully under normal operation to let the light in or closed to protect the glass or keep heat inside the room at night.

The louvers are opened and closed by a small computer-controlled motor, with two limit switches that sense when the skylight is fully open or fully closed. To open the skylight, one runs the motor in the forward direction until the fully open limit switch is activated. To close the skylight, one runs the motor similarly in

**Problem scope:** Write a program to control a motor that opens and closes a skylight.

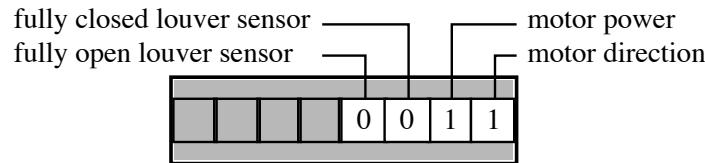
**Output required:** To the screen, a menu that will allow a user to select whether to open the skylight, close it, or report on its current position. To the motor controller, signals to start or stop the motor.

**Input:** The program receives menu selections from the user. It also receives status codes directly from the motor controller via the memory-mapped address that is its interface to the controller.

**Constants:** The memory-mapped addresses used by the multifunction chip in this program are `0xfffff7100` for the DR and `0xfffff7101` for the DDR. Bit positions in the DR, which follow, are numbered starting from the right end of the byte.

Bit #	In or Out?	Purpose	Settings	
0	Output	Motor direction	0 = forward	1 = reverse
1	Output	Motor power	0 = off	1 = on
2	Input	Fully closed louver sensor	0 = not fully closed	1 = fully closed
3	Input	Fully open louver sensor	0 = not fully open	1 = fully open

Data Register (DR): motor is on and louver is partly closed



Data Direction Register (DDR)

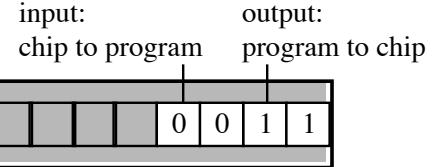


Figure 15.25. Problem specifications: Skylight controller.

the reverse direction. To know the current location of the skylight, one simply examines the state of the limit switches.

The motor is controlled by a box with relays and other circuitry for selecting its direction, turning it on and off, and sensing the state of the limit switches. The controller box has an interface to the computer through a multifunction chip using a technique known as *memory-mapped I/O*. This means that when certain main memory addresses are referenced, bits are written to or read from the multifunction chip, rather than real, physical memory.

In this program, we assume that the multifunction chip interfaces with the computer through two memory addresses: `0xfffff7100` refers to an 8-bit data register (DR) and `0xfffff7101` refers to an 8-bit data direction register (DDR). Each bit of the data register can be used to send data either from the hardware to the program, or vice versa. This is controlled by setting the DDR. Data flows from chip to program through a bit of the data register if the corresponding bit of the data direction register is 0. It flows from program to chip if the corresponding DDR bit is 1. The direction of communication, for each bit in the data register, is shown in Figure 15.25.

The program for the skylight controller consists of two parts<sup>7</sup>:

- A set of type declarations (Figure 15.26) and function prototypes.
- A main module containing:
  - A set of constants, initializers, and pointers that connect to the hardware addresses of the two registers.
  - The `main()` function (Figure 15.27) that initializes the system, displays a menu in an infinite loop, and waits for a user to request some service.

<sup>7</sup>The program was debugged using a device simulator, also written in C++ and using threads.

These declarations are stored in the file `skylight.h`, which is used in Figures 15.27 and 15.28.

```
#include <iostream>
using namespace std;

// Definitions of the registers on the multifunction chip
struct RegByte {
    unsigned int :4;
    unsigned int fullyOpen :1;
    unsigned int fullyClosed :1;
    unsigned int motorPower :1;
    unsigned int motorDirection :1;
};

typedef RegByte* DevicePointer;

// Definitions of the codes for the multifunction chip
enum PowerValues { motorOff = 0, motorOn = 1 };
enum DirectionValues { motorForward = 0, motorReverse = 1 };
enum Position { fullyClosed, partOpen, fullyOpen } ;

// Prototypes for the control operations.
Position skylightStatus( void );
void openSkylight( void );
void closeSkylight( void );
```

**Figure 15.26.** Declarations for the skylight controller: `skylight.hpp`.

- Three functions (Figure 15.28) that perform those services: opening the skylight, closing it, and asking about its current state.

#### Notes on Figure 15.26. Declarations for the skylight controller: `skylight.h`.

##### *First box: the bitfield type.*

- We use four bits to communicate with the multifunction chip; two are used by the program to receive status information from the chip and two are used to send control instructions to the chip. As shown in the register diagram, the leftmost four bits in the chip registers will not be used in this application. Therefore, the bitfield type declaration used to model a register begins with an unnamed field for the four padding bits, followed by named fields for two status bits and two control bits.
- We use pointer variables in the program to hold the addresses of the chip registers, so we declare a pointer type that references a register byte.

##### *Second box: status and switch codes.*

- Throughout the code, we could deal with 1 and 0 settings, as specified in Figure 15.25. However, doing so makes the code error prone and very hard to understand. Instead, we define three enumerated types to give symbolic names to the various switch settings and status codes. An array of strings is defined in Figure 15.27 parallel to the third enumeration, to allow easy output of the device’s status.
- Two of the enumerations are used simply to give names to codes. A series of `#define` commands could be used for this purpose, but `enum` is better because it is shorter and it lets us group the codes into sets of related values. We do not intend to use them as variable or parameter types. It is not necessary to set the symbols equal to 0 and 1 here because those are the default values in an enumeration. However it does make the correspondence between the symbols and the values clear and obvious.

**Third box: prototypes.** This package is composed of a main program and three functions used to carry out the user’s menu commands.

- The functions `openSkylight()` and `closeSkylight` are used to send commands to the device controller.
- The `skylightStatus` function is used to read the status bits set by the controller and determine whether it is an appropriate time to issue a new action command.

#### Notes on Figure 15.27. A skylight controller.

This program implements the specification of Figure 15.25. It uses the declarations in Figure 15.26 and calls functions in Figure 15.28.

```
#include "skylight.hpp"

// Attach to hardware addresses.
volatile DevicePointer const DR = (DevicePointer)0xfffff7100;
volatile DevicePointer const DDR = (DevicePointer)0xfffff7101;
// Create initialization constants.
const RegByte DDRmask = {0,0,1,1}; // select output bits
const RegByte DRinit = {0,0,0,0}; // 1 means program -> chip.
// Output strings corresponding to enum constants
const char* enumLabels[] = {"fully closed", "partially open", "fully open"};

int main( void )
{
    char choice;
    const char* menu = " O: Open skylight\n C: Close skylight\n"
                      " R: Report on position\n Q: Quit\n";
    // Initialize chip registers used by application.-----
    *DDR = DDRmask; // Designate input and output bits.
    *DR = DRinit; // Make sure motor is turned off.

    for (;;) {
        cout << menu << "Select operation: ";
        cin >> choice;
        choice = toupper( choice );
        if (choice == 'Q') break;
        switch (choice) {
            case 'O': openSkylight(); break;
            case 'C': closeSkylight(); break;
            case 'R': cout << "Louver is " << enumLabels[ skylightStatus() ] << "\n"; break;
            default: cout << "Incorrect choice, try again.\n";
        }
    }
    puts( " Skylight controller terminated.\n" );
    return 0;
}
```

Figure 15.27. A skylight controller.

***First box: setting up the registers.***

- We want a pointer variable `DR` to point at the address of the data register byte on the multifunction chip. We write its memory-mapped address as a hex literal, cast it to the appropriate pointer type, and store it in `DR`. The keyword `const` after the type name means that `DR` always points at this location and can never be changed.
- The keyword `volatile` means that something outside the program (in this case, the hardware device) may change the value of this variable at unpredictable times. We supply this information to the C++ compiler so that its code optimizer does not eliminate any assignments to or reads from the location that, otherwise, would appear to be redundant.
- Similarly, we set a pointer variable `DDR` to the address of the data direction register byte.
- A bitfield object, `DDRmask`, is created and will be used by the hardware device when the program is started, to define the communication protocol between device and program. The leftmost two bits are 0 to indicate that they will be used by the program to receive information from the device controller. The last two bits are 1, indicating that the program will use those positions to send control information to the chip.
- Another bitfield object, `DRinit`, is created for initializing the Data Register to an off state.

***Second box: chip initialization.***

- When the system is first turned on, the value `{0,0,1,1}` must be stored in the rightmost bits of the chip's data direction register to indicate the direction of data flow through the bits of the data register. Then the data register can be used to initialize the motor to an off state with a 0 bit vector.
- These two assignments take the bitfield objects that have been prepared and use a single assignment to store them into the two registers of the device controller. Completing the initialization in a single assignment is important when dealing with concurrent devices (computer and controller).

***Third box, outer: an infinite loop.***

- The remainder of the program is an infinite loop that presents an operation menu to the user and waits for commands. The user can quit at any time (and thereby turn the system off) by choosing option `Q`.
- We display a menu and read a selection. Then we use `toupper()` so that the program recognizes both lower-case and upper-case inputs in the `switch` statement.

***Inner box: calling the controller functions.***

- The system supports three operations: open the skylight, close it, and report on its position. There is a menu option and a function to call for each. The open and close commands change the state of the skylight using the last two bits of the Data Register. The report function checks on that state in the first two bits of the Data Register and returns a position code that `main()` uses to index the array `enumLabels` and display the position for the user.
- The `switch` has a `default` case to handle input errors. This case is not necessary; without it, an error would just be ignored. However, a good human interface gives feedback after every interaction, and an explicit error comment is helpful.

**Notes on Figure 15.28. Operations for the skylight controller.*****First box: skylightStatus().***

- There are three possible states: fully open, fully closed, and somewhere in between (represented by both status bits being off). This information is vital to the program because it must turn off the motor when the skylight reaches either limit.
- The status is tested by checking the appropriate bitfields in `DR` to see which of the mutually exclusive conditions exists and then returning the corresponding position code to `main()`.
- Note that, throughout, we use the symbolic names of the bitfields and the symbolic enum constants to make the code comprehensible.

***Second box, outer:*** openSkylight().

- The basic steps of opening the skylight are turn on the motor, wait until the louvers are completely open, then turn off the motor.
- Sometimes it is not necessary to turn on the motor. If a command is given to open the skylight when it is already open, control simply returns to the menu.
- In between turning on the motor and turning it off, the program sits in a tight **while** loop, waiting for the open status bit in DR to change state due to the controller's actions. This technique is referred to as a *busy wait loop*, and is only appropriate for an embedded system.
- Turning the motor off is also done with a single assignment, using a prepared bitfield object.
- Second box, first inner box: turning on the motor.

These functions are called from Figure 15.27 and are in the same source code file.

```
// --- Return skylight position-----
position
skylightStatus( void )
{
    if (DR->fullyClosed) return fullyClosed;
    else if (DR->fullyOpen) return fullyOpen;
    else return partOpen;
}

// --- Open skylight -----
void
openSkylight( void )
{
    if (DR->fullyOpen) return; // Don't start motor if already open

    regByte dr = { 0, 0, motorOn, motorForward }; // Prepare start signal.
    *DR = dr; // Store start signal in hardware register

    while (!(DR->fullyOpen)); // Wait and watch until louvre is open

    dr.motorPower = motorOff; // Set control bit to off
    *DR = dr; // Store stop signal in hardware register
}

// --- Close skylight -----
void
close_skylight( void )
{
    if (DR->fullyClosed) return; // Don't start motor if already closed

    regByte dr = { 0, 0, motorOn, motorReverse }; // Prepare start signal.
    *DR = dr; // Store start signal in hardware register
    while (!(DR->fullyClosed)); // Wait and watch until louvre is closed

    dr.motorPower = motorOff; // Set control bit to off
    *DR = dr; // Store stop signal in hardware register
}
```

Figure 15.28. Operations for the skylight controller.

- If the louver needs to be opened, the program must turn on the motor in the forward direction. In this case, both the direction and power bits of the control byte must be changed.
- To do this, we prepare a local `RegByte` object with the right control bits.
- Then the contents of this byte are copied to `DR`, using a single assignment to initiate the motor’s action.
- In between turning on the motor and turning it off, the program sits in a tight `while` loop, waiting for the open status bit in `DR` to change state due to the controller’s actions<sup>8</sup>.
- Second inner box: turning off the motor. A similar pattern is repeated to turn off the motor. The motor power bit in the local `RegByte` is set to off and the byte is transferred to `DR`. The direction bit needs no adjusting since that bit is irrelevant when the motor is off.

**Last box:** `closeSkylight()`. The `closeSkylight()` function is exactly analogous to `openSkylight()`. It returns promptly if no work is needed. Otherwise, it turns the motor on in the reverse direction, waits for the `fullyClosed` bit to be turned on by the controller, then turns the motor off.

**Testing.** To run this program in reality, we would need an automated skylight, a motor, a board containing the electronics necessary to drive the motor, and a multifunction chip attached to a computer. However, code always should be tested, if possible, before using it to control a physical device. This program was tested using a skylight simulator: a separate program that emulates the role of the real device, supplies feedback to the program written in this section, and prints periodic reports for the programmer. (This simulation program is too complex to present here in detail.)

Output from a simulation follows. The initial state of the simulated skylight is half open (the result of a power failure). In the output, when you see a line that starts with `SKYLIGHT`, you should imagine that you hear a motor running and see the louvers changing position. To reduce the total amount of output, repetitions of the menu have been replaced by dashed lines.

```
SKYLIGHT: louver is 47% open
```

```
Select operation:
```

```
O: Open skylight
C: Close skylight
R: Report on position
Q: Quit
```

```
Enter letter of desired item: r
```

```
Louver is partially open
```

---

```
Enter letter of desired item: o
```

```
SKYLIGHT: louver is 60% open
SKYLIGHT: louver is 80% open
SKYLIGHT: louver is 100% open
SKYLIGHT: louver is 101% open
```

---

```
Enter letter of desired item: o
```

---

```
Enter letter of desired item: c
```

```
SKYLIGHT: louver is 100% open
SKYLIGHT: louver is 80% open
SKYLIGHT: louver is 60% open
SKYLIGHT: louver is 40% open
SKYLIGHT: louver is 20% open
SKYLIGHT: louver is 0% open
SKYLIGHT: louver is -1% open
```

---

<sup>8</sup>This technique is referred to as a busy wait loop, and is only appropriate for a dedicated computer or an embedded system.

Operator	Meaning and Use
<code>~</code>	Used to toggle bit values in vector.
<code> </code>	Used to combine sections of a vector or turn on bits.
<code>&amp;</code>	Used to isolate sections of a vector or turn off bits.
<code>~</code>	Used to compare two bit vectors or toggle bits on and off.
<code>&lt;&lt;</code>	Used to shift bits left; fills right end with 0 bits. Shifting an integer left by one bit has the same effect as multiplying it by 2.
<code>&gt;&gt;</code>	Used to shift bits right. Unsigned right shift fills left end with 0 bits; signed right shift fills left end with copies of the sign bit. Shifting an integer right by one bit has the same effect as dividing it by 2.

Figure 15.29. Summary of bitwise operators.

```
Enter letter of desired item: R
Louver is fully closed
-----
Enter letter of desired item: q
Skylight controller terminated.
```

Note that the motor always “overshoots” by a small amount. This happens because it takes a brief time to turn off the motor after the sensor says that the louver is fully open or fully closed. A real device would have to be calibrated with this in mind.

## 15.6 What You Should Remember

### 15.6.1 Major Concepts

**Unsigned numbers.** At times, applications naturally use numbers whose range always is positive, such as memory addresses and the pixel values of digital images. In these cases, using an `unsigned` type allows for larger ranges and helps prevent errors during calculations.

**Hexadecimal.** C supports input, output, and literals for hexadecimal integers. Hex notation is important because it translates directly to binary notation and, therefore, lets us work easily with bit patterns.

**I/O conversions.** We use C formatted output here because it is easier to control the spacing, field length, and leading zeroes in C than it is with C++. All built-in data types have format conversion specifiers. Unsigned data values are displayed using `%u` in decimal form and `%x` in hexadecimal form. For any data type, it is possible to specify a leading fill character other than blank. This is useful for printing leading zeroes in hex values, as in `%08x`.

**Bit arrays** The bits in an integer data value can be viewed as an array and manipulated using bitmasks and bitwise operators.

**Bitmasks.** A mask is used to specify which bit or bits within an integer are to be selected for an operation. To create a mask, first write the appropriate bit pattern in binary, then translate it to hexadecimal and `#define` the result as a constant.

**Summary of bitwise operators.** Bitwise operations are used with bitmasks expressed in hexadecimal notation to operate on both single bits and groups of bits within an integer. Figure 15.29 is a summary of these operators.

**Bitfields.** Bitfield structures provide a symbolic alternative to the manual manipulation of bits using the bitwise operators. A structure can be declared with components that are smaller than a byte, and those components can be initialized like the fields of an ordinary structure and used like ordinary integers with small ranges. The compiler does all the shifting and masking needed to access the separate bits.

**Compression and encryption.** Information is compressed to save storage space and transmission time, and/or encrypted to preserve privacy. These applications make extensive use of bitwise operations.

**Volatile.** When used in a variable declaration, this type qualifier indicates that an outside agent such as a hardware device may change the variable's value at any time. Using the qualifier instructs the compiler not to attempt to optimize the use of the variable.

### 15.6.2 Programming Style

**Constants.** As with other constants, use `#define` for bitmasks to generate more understandable code.

**Hexadecimal vs. decimal.** It is important to *understand* numbers written in hexadecimal, binary, and decimal notations. Since we cannot use binary constants in the code, we need to know when to *use* hexadecimal and when to use decimal notation. Use hexadecimal for literals when working with bitwise operators or interacting with hardware devices because the pattern of bits in the number is important and the translation from hex to binary is direct. Use decimal notation for most other purposes.

**Proper data type.** In addition to using the appropriate literal form, the proper integer data type should be chosen. The range of needed values should help you to decide between signed and unsigned, as well as the shortest length necessary.

**Bit operators vs. bitfields.** One must choose whether to use bit operators or bitfield structures as a representation. Bitfield structures are appropriate for describing the bit layout of a hardware device interface that is unlikely to change. They enable us to use field names instead of bit positions to interact with the device more easily. When dealing with a series of bytes from a stream, it is more convenient to use unsigned integers and the bitwise operations, as with the encoding and decoding of information.

### 15.6.3 Sticky Points and Common Errors

- Some pocket calculators provide a way to enter a number in base 10 and read it out in base 16 or vice versa. If yours does not, use the decimal to binary to hexadecimal method presented in Appendix E to do number conversions. The algorithm that converts directly from base 10 to base 16 relies on modular arithmetic, which is not supported by many calculators.
- When generating bitmasks beware of inverting the bit pattern. Make sure the ones and zeroes are appropriate for the bitwise operation being performed.
- Do not confuse the logical operators `&&`, `||`, and `!` with the bitwise operators `&`, `|`, and `~`. Logical operators do whole-word operations and return only the values `true` or `false`. Bitwise operators use bit vectors of varying length as both input and output.
- Do not confuse the use of `&` and `|` in an expression. Keep straight whether you are using the mask to select bits from a value or to introduce bits into it.
- Beware of shifting a bit vector in the wrong direction, either by using the incorrect operator or mistakenly using a negative shift amount.
- Remember that the data type determines whether a signed bit extension or unsigned zero-fill is performed during right shifts.
- Beware of the relative precedence of the bitwise operators. There are many different precedences, and not all seem natural to someone unfamiliar with manipulating bits.
- Beware of being off by one bit position. It is quite common when shifting or manipulating a particular bit to miss and be off by one position. Knowing the powers of 2 can help reduce the number of times this happens.
- When using bitfields that have lengths shorter than a byte, don't try to store values that are too long to be represented. The most significant bits will be truncated, leading to wrong and confusing results.

### 15.6.4 New and Revisited Vocabulary

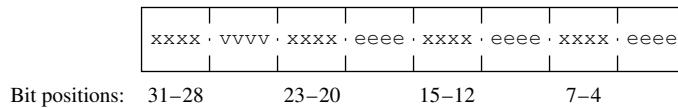
These are the most important terms, concepts, keywords, and operators discussed in this chapter.

binary	$\sim$ (complement)	bit vector
decimal	$<<$ (left shift)	mask
hexadecimal	$>>$ (right shift)	toggle
unsigned integer	signed shift	unpack and recombine
hex integer literal	unsigned shift	encode and decode
hex character literal	$\&$ (bitwise AND)	encryption
<code>%x</code> (hex I/O conversion)	$\sim$ (bitwise exclusive OR)	bitfield structure
<code>%u</code> (unsigned conversion)	$ $ (bitwise OR)	<b>volatile</b>
two's complement	portable notation	memory-mapped I/O

## 15.7 Exercises

### 15.7.1 Self-Test Exercises

1.
  - (a) What decimal numbers do you need to convert to get the following hexadecimal outputs: AAA, FAD, F1F1, and B00 ?
  - (b) Write the decimal numbers from 15 to 35 in hexadecimal notation.
  - (c) Write the decimal value 105 in hexadecimal notation.
  - (d) What hexadecimal number equals 101010 in binary?
  - (e) Write any string of 16 bits. Show how you convert this to a hexadecimal number. Then write this number as a legal C hexadecimal literal.
2. Compile and run the program in Figure 15.10. Experiment with different input and answer the following questions:
  - (a) What happens if you try to print a negative signed integer with a `%hu` format?
  - (b) What is the biggest `unsigned int` you can read and write? Note what it looks like in hexadecimal notation; you easily can see why this is the biggest representable integer. Explain it in your own words.
  - (c) You can read numbers in hexadecimal format. Try this. Enter a series of numbers and experimentally find one that prints the value  $117_{10}$ .
3. As part of a cryptographic project, 4-byte words are scrambled by moving around groups of 4 bits each. A scrambling pattern follows: it has three sets of bits labeled with the letters `x`, `v`, and `e`. The bits in the `x` positions do not move. All the `e` bits move 8 bits to the left, and the `v` bits go into bits 3...0.



- (a) Define three AND masks to isolate the sets of fields `x`, `v`, and `e`.
  - (b) Write three AND instructions to decompose the word by isolating the three sets of bits and storing them in variables named `X`, `V`, and `E`.
  - (c) Write the two shift instructions required to reposition the bits of `E` and `V`.
  - (d) Write an OR instruction that puts the three parts back together.
4. A college professor is writing a program that emulates the computer he first used in 1960 (an IBM 704). He wants to use this program to teach others about the way in which hardware design has progressed in 40 years. The instructions on this machine were 36 bits long and had four parts:

- |                      |            |
|----------------------|------------|
| (a) Operation code   | bits 35–33 |
| Decrement field      | bits 32–18 |
| Index register field | bits 17–15 |
| Address field        | bits 14–0  |

op	decrement	idx	address
bit positions: 35-33	32-18	17-15	14-0

In this machine, the decrement field is used as part of the operation code in some instructions and as a second address field in others. Write a `typedef` and bitfield structure declaration to embed this instruction layout in 6 bytes. Put the necessary padding bits anywhere convenient but make sure that the two 3-bit fields do not cross byte boundaries and the two longer fields cross only one boundary each.

5. Repeat problem 4 using bit masks. Assume that the op and index fields are stored together in one byte and the other fields occupy two bytes each.
6. What is stored in `w` by each line of the following statements? Write out the 8-bit vectors for each variable and show the steps of the operations. Use these values for the variables:

```
unsigned char v, w, x = 1, y = 2, z = 4;
const unsigned char mask = 0xC3;

a. w = ~x & x; f. w = ~x | x;
b. w = ~x ~ x; g. w = 1; w<<= 1;
c. w = x | y & x | z; h. w = x ~ ~ y;
d. v = y | z; w = (v<<3) + (v<<1); i. w = x | y | z>>2;
e. w = x | y & x | z<<y ~ mask>>x; j. w = x & y & ~ z;
```

### 15.7.2 Using Pencil and Paper

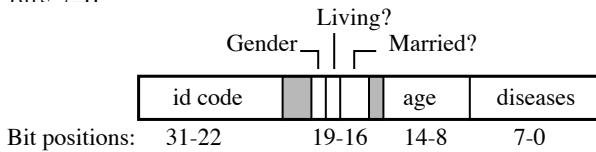
1. (a) Write your full name, including blanks, in the ASCII character code in hexadecimal literal form.  
(b) What base-10 number equals the two's complement binary value 11010110 ?  
(c) Do the following addition in binary two's complement: 01110110 + 10001001 = ?  
(d) Express 00110110<sub>2</sub> in bases 10 and 16.  
(e) Express 5174<sub>10</sub> in binary and hexadecimal.  
(f) Use binary arithmetic to multiply 00010110<sub>2</sub> by 4<sub>10</sub>, then express the answer in bases 16 and 10.  
(g) How many times would you need to divide 100110<sub>2</sub> by 2<sub>10</sub> to get the quotient 1?
2. As part of a cryptographic project, every 4 bytes of information in a data stream will be scrambled. One of the scrambling patterns follows: it has five fields labeled with four letters. The bits in the positions labeled `x` are not moved. The bit field labeled `n` swaps places with the field labeled `v`; the rightmost bit of `n` moves to position 5 and the leftmost bit of `v` shifts to position 27. The field labeled `e` moves 2 bits to the left to make space for this swap.

xxxx	nnnnnnnnnnnn	eee	vvvvvvvv	xxxxx
Bit positions: 31-28	27-17	16-14	13-5	4-0

- (a) Define four AND masks to isolate the fields `x`, `n`, `e`, and `v`.  
(b) Write four AND instructions to decompose the word, isolating the four sets of bits and storing them in variables named `X`, `N`, `E`, and `V`.  
(c) Write the three shift instructions required to reposition the bits of `N`, `E`, and `V`.  
(d) Write the OR instruction to put the four parts back together.

3. Repeat parts (a) and (b) of exercise 2 using a bitfield structure. Explain why parts (c) and (d) are easier to do using masks than using bitfields.
4. (a) Write the numbers from  $30_{10}$  to  $35_{10}$  in binary notation.  
 (b) Write  $762_{10}$  in hexadecimal notation.  
 (c) Write  $762_{10}$  in binary notation.  
 (d) What hexadecimal number equals  $11000011_2$ ?  
 (e) What base-10 number equals  $11000011_2$ ?  
 (f) What decimal numbers equal the following hex “words”: C3b0, dab, add, and feed?
5. (Advanced topic.) The following three riddles have answers related to the limitations of a computer and the quirks of the C language in representing and using the various integer types.
  - (a) When is  $x + 1 < x$ ?
  - (b) When does a positive number have a negative sign?
  - (c) When is a big number small?
6. You are writing a program to analyze health statistics. A data set containing records for a small village has been collected, coded, and stored in a binary file. The data cover about 1,000 residents. Each person’s data are packed into a single long unsigned integer, with fields (shown in the diagram) defined as follows:
 

(a) ID code	bits 31–22	
Gender	bit 19	0 = male, 1 = female
Alive	bit 18	0 = dead, 1 = living
Marital status	bits 17–16	0 = single, 1 = married, 2 = divorced, 3 = widowed
Age	bits 14–8	
Disease code	bits 7–0	



Bit positions: 31-22      19-16      14-8      7-0

The remaining bit positions (gray in the diagram) are irrelevant to the current research and may be ignored. Write a `typedef` and bitfield structure declaration to describe these data.

7. Solve for N in the following equations. Show your work, using an 8-bit unsigned binary representation for the constants and the variables N and T. Give the answer to each item in binary and hexadecimal format.
 

a. $N = 21_{16} \mid 39_{16}$	e. $N = 2A_{16} \& 0F_{16}$
b. $3A_{16} \sim N = 00_{16}$	f. $N = 4C_{16} \ll 3_{16}$
c. $N = \sim AE_{16} \mid 39_{16}$	g. $N = A1_{16} \& 39_{16} \mid 1E_{16}$
d. $N = 4C_{16} \gg 1$	h. $T = 2A_{16}; \sim T \gg 2 \sim N = 01_{16}$

### 15.7.3 Using the Computer

1. The case bit.

In an alphabetic ASCII character, the third bit from the left is the *case-shift bit*. This bit is 0 for upper-case letters and 1 for lower-case letters. Except for this bit, the code of an upper-case letter is the same as its lower-case counterpart. Write a program that contains a bitmask for the case-shift bit. Read input from the keyboard using `getline( cin, line)`, where `line` is a string variable. Do the operations described below, but make sure to change only alphabetic letters; do not alter the values of numerals, special symbols, and the like. Use the library function `isalpha()` to test the chars.

Assume that each line, including the last, will end with a newline character.

- (a) On the first line of input, use the mask and one bitwise operator to toggle the case of the input; that is, change B to b or b to B. Print the results.
  - (b) On the second line of input, use your mask and one bitwise operator to change the input to all uppercase. Print the results.
  - (c) On the third line of input, use your mask and one bitwise operator to change the input to all lowercase. Print the results.
2. Decomposing an integer.

One of the fastest methods of sorting is a *radix sort*. To use this algorithm, each item being sorted is decomposed into fields and the data set is sorted on each field in succession. The speed of the sort depends on the number of fields used; the space used by the sorting array depends on the number of bits in each field. A good compromise for sorting 32-bit numbers is to use four fields of 8 bits each, as in the following diagram. During the sorting process, the data are sorted first based on the least significant field (A, in the diagram) and last using the most significant field (D).



This problem focuses on only the decomposition, not the sorting. Assume you want to sort long integers and need to decompose each into four 8-bit portions, as shown in the diagram. Write a function, `getKey()`, that has two parameters: the number to be decomposed and the pass number, 0...3. Isolate and return the field of the number that is the right key for the specified pass. For pass 0, return field A; for pass 1, field B; and so forth. Write a main program that will let you test your `getKey()` function. For each data value, echo the input in both decimal and hexadecimal form, then call the `getKey()` function four times, printing the four fields in both decimal and hexadecimal format.

3. A computational mystery.

- (a) Enter the following code for the `mystery()` function into a file and write a main program that will call it repeatedly. The main program should enter a pair of numbers, one real and one positive integer; call the `mystery()` function; and print the result.

```
double mystery( double x, int n )/ Assume n >= 0
{
    double y = 1.0;
    while (n > 0) {
        if (n & 1) y *= x;      // Test if n is odd.
        n >>= 1;
        x *= x;                // Square x.
    }
    return y;
}
```

- (b) Manually trace the execution of the code for `mystery()`. Use the argument values `n = 5` and `x = 3.0`, then try again with `x = 2.0` and `n = 6`. On a trace chart, make columns for `x`, `y`, `n`, `n` in binary, and `n & 1`. Show how the values of each expression change every time around the loop.
  - (c) Run the program, testing it with several pairs of small numbers. Chart the results. The `mystery()` function uses an unusual method to compute a useful and common mathematical function. Which mathematical function is it?
  - (d) Turn in the program, some output, the chart of output results, the hand emulation, and a description of the mathematical function.
4. Binary to decimal conversion.

Appendix E, Figure E.4 describes and illustrates one method for converting binary numbers to base 10. Write a program to implement this process, as follows:

- (a) Write a function, `strip()`, that returns the value of the rightmost bit of its `long` integer argument. This will be either 0 (`false`) or 1 (`true`). Implement this using bit operations.

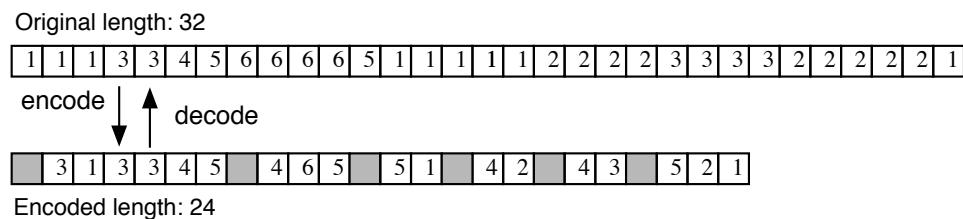
- (b) Write a `void` function, `convert()`, that has one `long signed` integer argument. It should start by printing a left parenthesis and finish by printing a matching right parenthesis. Between generating these, repeatedly strip a bit off the right end of the argument (or what remains of it). If the bit is a 1, print the corresponding power of 2 and a `+` sign. Then shift the argument 1 bit to the right. Continue to test, print, and strip until the remaining argument is 0.
- (c) Write a main program to test the conversion function, making it easy to test several data values. For each, prompt the user for an integer. If it is negative, print a negative sign and take its absolute value. Then call the `convert()` function to print the binary value of the number in its expanded form. For example, if the input was `-25`, the output should be `-(1 + 8 + 16)`.
5. Input hexadecimal, output binary.  
 Appendix E, Figure E.5 describes and illustrates a method for converting hexadecimal numbers to binary. Write a program to implement this process, as follows:
- Define a constant array of `char*` to implement the bit patterns in the third column of the table in Figure E.5.
  - Write a function, `strip()`, that has a call-by-address parameter, `np`, the number being converted. Use a masking operation to isolate the rightmost 4 bits of `np` and save this value. Then shift `np` 4 bits to the right and return this modified value through the parameter. Return the isolated hexadecimal digit as the value of the function.
  - Write a `void` function, `convert()`, that has one `long signed` integer argument, `n`, and an empty integer array in which to store an answer. Remember that integers are represented in binary in the computer and 4 binary bits represent the same integer as 1 hexadecimal digit. Repeatedly strip a field of 4 bits off the right end of the argument (or what remains of it) and store the resulting hexadecimal digit in successive slots of the answer array. Do this eight times for a 32-bit integer.
  - Write a main program to test your conversion function, making it easy to test several data values. For each, prompt the user to enter a hexadecimal integer in the form `10A2`, and read it using a `%x` format specifier. Then call the `convert()` function, sending in the value of the number and an array to hold the hexadecimal expansion of that number. After conversion, print the number in expanded binary form. Start by printing a left parenthesis, then print the eight hexadecimal digits of the number in their binary form, using the array from part (a). Finish by printing a matching right parenthesis. For example, if the hexadecimal numbers `-1` and `17` were entered, the output should be:  

$$\begin{array}{l} ( \text{1111 } ) \\ ( \text{0000 } \text{0000 } \text{0000 } \text{0000 } \text{0000 } \text{0000 } \text{0001 } \text{0111 } ) \end{array}$$

6. Compressing an image—run length encoding.

Digital images often are large and consume vast amounts of disk space. Therefore, it can be desirable to compress them and save space. One simple compression method is *run length encoding*. This technique is based on the fact that most images have large areas of constant intensity. A series of three or more consecutive pixels of the same value is called a *run*. When compressing a file, any value in the input sequence that is not part of a run simply is transferred from the original sequence into the encoded version. When three or more consecutive values are the same, an encoding triplet is created. The first element of this triplet is an escape code, actually a character value not expected to be present in the data. The second value is the number of characters present in the run, stored as a 1-byte unsigned integer. The last value is the input value that is repeated the indicated number of times.

The compression process is illustrated below. A stream of characters is shown first in its original form, and below that in compressed form. The escape code characters are shown as gray boxes.

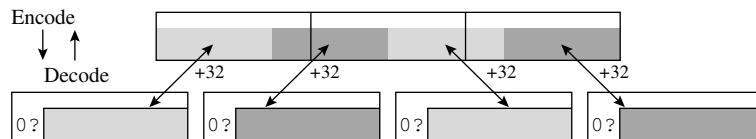


When a compressed image is used, the encoding process must be reversed. All encoding triplets are expanded into a corresponding sequence of identical byte values.

Write a program that will perform run length encoding and decoding. First, open a user-specified file for input (appropriately handle any errors in the file name) and another file for output. Then ask the user whether encoding or decoding should be performed. If encoding, read successive bytes from the input stream, noting any consecutive repetitions. Those occurring once or twice should be echoed to the output file, while runs of length three or more should be converted into an encoding triplet, which then is written to the output file. Use a value of 0xFF (255) as the escape code for this problem. If a data value of 255 occurs, change it to 254. Any runs in the data longer than 255 should be broken into an appropriate number of triplets of that length, followed by one of lesser length to finish the description. If decoding, read successive bytes from the input stream, scanning for encoding triplets. All characters between these triplets should be copied directly into the output file. Encoding triplets should be expanded to the appropriate length and sent to the output file.

#### 7. Encoding a file—uuencode.

One command in the **UNIX** operating system is *uuencode*, and its purpose is to encode binary data files in **ASCII** format for easy transmission between machines. The program will read the contents of one data file and generate another, taking successive groups of three input bytes and transforming them into corresponding groups of four **ASCII** bytes as follows:



The original 3 bytes (24 bits) are divided into four 6-bit segments. The value of each segment is used to generate a single byte of data. The value of this byte is calculated by adding the value 32 to the 6-bit value and storing the result in a character variable. This creates a printable **ASCII** value that can be transmitted easily. In the event that the original file length is not divisible by 3, fill in the missing bytes at the end by bytes containing the value 0. The final byte triplet then is converted in the same manner.<sup>9</sup>

Write a program that will perform the uuencode operation. First, open a file with a user-specified name for input (appropriately handle any errors in the file name). Open another file for output. Then read successive byte triplets from the input stream and write corresponding 4-byte chunks to the output stream, adding 0 bytes at the end, if necessary, to make 3 bytes in the final triplet.

#### 8. Decoding a file—uudecode.

The matching command to the uuencode command discussed in the previous exercise is uudecode. It takes as input an **ASCII** file created using uuencode and produces the corresponding binary file. It simply reverses the encoding process just described, taking 4-byte groups as input and generating 3-byte groups as output. Write a program that will perform the uudecode operation. First, open a file with a user-supplied name for input (appropriately handle any errors in the file name). Open an output file as well. Perform the conversion process by transforming successive 4-byte input groupings into 3-byte output groups and writing them to the output file.

#### 9. Set operations.

A set is a data structure for representing a group of items. The total collection of items that can exist in a set is referred to as the *set's domain*. A set with no items in it is called an *empty set*, while one that contains all the items is called a *universal set*. Items can be added to a set, but if that item already exists in the set, nothing changes. Items also can be removed from the set. In addition, some operations create new sets: (1) *intersection* produces the set of items common to two sets, (2) *union* produces the set containing all items found in either of two sets, (3) *difference* produces the set containing those items in a first set that do not exist in a second set, and (4) *complement* produces the set containing those items in the domain that are not present in a set.

There are many ways to represent a set in C. In this exercise, define a set as a bit vector. Represent the domain of the set as an enumeration. Each bit position in the vector corresponds to an item in the domain. A bit value is 1 if that item is present in the set. Each of the various set operations can be performed by doing an appropriate bitwise operation on sets represented in this manner. Write a program that allows a user to define and manipulate sets in the following manner:

<sup>9</sup>The real uuencode command also generates header and trailer information that is useful in decoding the file. For simplicity, we ignore this information in the exercise. Newer versions add 96 (not 32) to the null character.

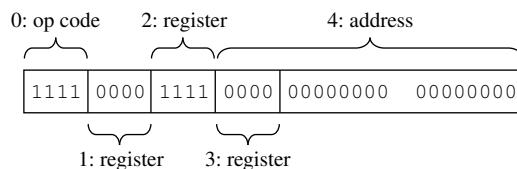
Machine Instruction	Op Code	1st Register	2nd Register	3rd Register	Address	Description
Load	0000	R	—	—	M	Load R with contents of M
Store	0001	R	—	—	M	Store contents of R into M
Add	0010	Ri	Rj	Rk	—	Rk = Ri + Rj
Subtract	0011	Ri	Rj	Rk	—	Rk = Ri - Rj
Multiply	0100	Ri	Rj	Rk	—	Rk = Ri * Rj
Divide	0101	Ri	Rj	Rk	—	Rk = Ri / Rj
Negate	0110	Ri	Rj	—	—	Rj = -Ri
AND	0111	Ri	Rj	Rk	—	Rk = Ri & Rj
OR	1000	Ri	Rj	Rk	—	Rk = Ri   Rj
XOR	1001	Ri	Rj	Rk	—	Rk = Ri ~ Rj
NOT	1010	Ri	Rj	—	—	Rj = ~Ri
<	1011	Ri	Rj	Rk	—	Rk = Ri < Rj
>	1100	Ri	Rj	Rk	—	Rk = Ri > Rj
=	1101	Ri	Rj	Rk	—	Rk = Ri == Rj
Branch U	1110	—	—	—	M	Branch to address M
Branch C	1111	R	—	—	M	Branch to M if R true

Figure 15.30. Problem 10.

- Let the domain be the colors red, orange, yellow, green, blue, purple, white, and black. To map from a color to a bit in the set representation, start with a bit vector containing the integer value 1 and use the enumeration value as a left shift amount.
- Write a group of functions that will (a) add an element to a set, (b) remove an element from a set, (c) test if an element is in the set, and (d) print the contents of a set. These will be support functions for the rest of the program.
- Write a group of functions that perform set intersection, set union, set difference, and set complement.
- Write a main program that presents an initial menu to the user, the options being the four set operations just mentioned. When one of these options is chosen, the program should ask the user to supply the contents of the necessary sets, perform the operation, and print the contents of the resulting set. To supply the contents of a set, the user should be able to add and remove items from the set repeatedly until the desired group has been produced. When adding or removing an item, the user should be able to type the name of the item (a color) or select it from a menu and the program then should adjust the representation of the set accordingly.

#### 10. Machine code disassembler.

The last phase of the compilation process is turning assembly language into machine language. Sometimes it is desirable to reverse this process by generating assembly language from machine code, or *disassembling*. This is similar to what happens in the instruction decoding step of the CPU. We describe a fictitious machine that has 16 assembly language instructions encoded in a 4-byte machine instruction. This 4-byte area can be decomposed into five pieces. The leftmost half (2 bytes) is split into 4-bit fields. The remaining half may be used in conjunction with the last 4-bit field from the left half to form another large field, as follows:



The contents of these fields depend on the particular instruction, described in Figure 15.30. The operation code determines whether there is a third register number or those bits are to be used as part of a memory address. Write a program that prints the assembly language program described by a machine code file.

First, open a user-specified binary file for input (appropriately handle any errors in the file name). Read successive groups of 4 bytes and treat each group as an instruction. Decode the instruction based on looking at the op code field and interpreting the remaining fields properly. Print an assembly language line describing this. Print register numbers as R1 or R15. Print the 20-bit memory addresses in hexadecimal form. For instance, the two machine instructions on the left might be translated into the assembly instructions on the right:

```
00000101 00001111 11001101 01100100 → LOAD R5, 0xFCD64  
00101101 00101011 00000000 00000000 → ADD R13, R2, R11
```

# **Part V**

# **Pointers**



# Chapter 16

## Dynamic Arrays

In this chapter, we introduce two important parts of the C++ language: allocating arrays dynamically and using pointers to process them. We show how pointers and subscripts are alternative ways to process an array, how an array of unpredictable length can be allocated at run time, and how dynamic allocation can be used to remove limitations from algorithms that work with arrays. Destructors are introduced for managing the dynamic memory.

Finally, two basic sorting algorithms are presented and used to illustrate all the other techniques introduced in the chapter.

### 16.1 Operator Definitions

In C++, a programmer may define additional methods for any operator defined by the C precedence table<sup>1</sup>. We call these definitions operator extensions, and they are a very powerful tool for making it possible to write C++ code that looks like the formulas you would see in a textbook on mathematics or physics. For example, it is common to define the arithmetic operators on the class Complex.

An operator definition must have the same number of parameters as the original C operator, and it will have the same precedence and associativity. If it also has the same purpose and general meaning as the original, we call it an operator extension. For example, if the definition of + on Complex performs complex addition, then the new operator is an extension of the original.

If the new operator has a different meaning, it is generally called an operator overload. For instance, if you define + to mean string concatenation, that is an overload, because now + has two unrelated meanings. Operator overloads should normally be avoided, although some are built into C++, for example << and >> for I/O and + for strings.

Operator extensions should be used carefully, and only to allow a new class to “behave like” the programmer expects it to behave. For example, in this chapter we define a class, Flex, which implements an array that can grow longer if needed to store an unpredictable amount of information. Because this class is used to replace simple C arrays, programmers want it to behave like an array – they want to be able to access the data inside the Flex by using subscript. So the Flex class *extends* the subscript operator, and delegates the subscripting operation to the simple dynamic array *inside* the Flex array. You can see the prototype for the new subscript function in Figure 16.11 and the actual method definition in Figure 16.12.

### 16.2 Pointers—Old and New Ideas

This section collects, reviews, and elaborates on the material concerning pointers introduced in earlier Chapters, and extends it to show new ways pointers may be used with arrays. In many ways, a pointer in C++ is like a pronoun in English. Both can be attached, or bound, to an object and changed later to refer to a different object.

---

<sup>1</sup>A complete treatment of this topic is beyond the scope of this book. However, in the next few chapters, we need to be able to extend the subscript operator, so that one special operator extension will be presented.

We declare an array and show how to set a pointer to its first slot or to an interior slot.

```
double z[3] = {0.0, 1.0, 0.0};
double* zp = z;
double* wp = &z[2];
```

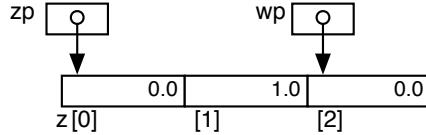


Figure 16.1. Pointing at an array.

### 16.2.1 Pointer Declarations and Initialization

A pointer variable is created when we use an asterisk after a type name in a declaration. The declaration `int k`, with no `*`, creates an `int` variable, but `int* p` creates a pointer variable that can refer to any `int`. The type named is called the **base type** of the pointer, and the pointer can point meaningfully only at variables of that type.

The **referent** of a pointer (the address of an object of a matching type) can be set either by initialization or by assignment. Figures ?? through ?? give examples of declarations of several types of pointers. In the following paragraphs, we briefly summarize the syntax and meaning of these pointer assignments.

**Pointing at an array.** In C and in C++, an array is a sequence of variables that have the same type and are stored consecutively and contiguously in memory. When we point at an array, we actually point only at one of the elements (slots) in that array, not at the entire object. However, pointing at one slot gives us access to all the other slots that precede and follow it. Figure 16.1 illustrates pointing to an array.

To point at a single slot in an array, we use the array name with both an ampersand and a subscript, as in `wp = &z[2]`. To set a pointer to the beginning of an array, we could write `zp = &z[0]` or, more simply, omit both the ampersand and subscript and write `zp = z`. We normally use the second form. This simpler syntax works because, in C and in C++, the name of an array is translated as a pointer to the first slot of that array.<sup>2</sup> Similarly, there are two ways to set a pointer to the third array element: `zp = &z[2]`, as just discussed, and `zp = z+2`, which is discussed in a later section.

### 16.2.2 Using Pointers

To use pointers skillfully, the meanings of several pointer operations must be understood. These include subscript, direct and indirect reference, direct and indirect assignment, input and output through pointers, and pointer arithmetic. The following paragraphs review or present these operations.

**Pointers with subscripts.** Syntactically, there is no difference between using an array name and a pointer to an array. If the referent of pointer `p` is one of the slots within an array, then `p` can be used with a subscript to refer to the other slots of that array. For example, in Figure 16.1, the pointer `zp` refers to array element `z[0]`, so using a subscript with `zp` means the same thing as using a subscript with `z`, and `&zp[1]` means the same thing as `&z[1]`. The subscript used with a pointer is interpreted relative to the pointer's referent. In Figure 16.1, pointer `wp` refers to `z[2]`, so `wp[0]` means the same thing as `z[2]`, and `wp[1]` would be the slot after the end of the diagrammed array. The ability to use a **pointer with a subscript** is important because array arguments are passed by address. Within a function, the parameter is a pointer to the beginning of the array argument. The notational equivalence of arrays and pointers lets us use subscripts with array parameters.

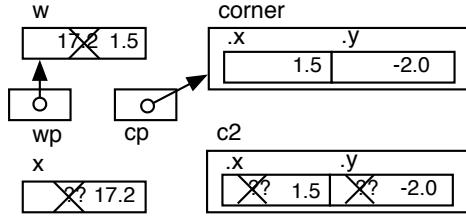
**Indirect reference.** Two operators in C++ dereference pointers: the asterisk (`*`) and the arrow (`->`). The expression `*p` stands for the referent of `p`, the same way a pronoun stands for a noun. We say that we can use `p` to reference `k` *indirectly*. If `p` points at `k`, then `m = *p` means the same thing as `m = k`. Longer expressions also can be written; anywhere that you might write a variable name, you can write an asterisk and the name of a pointer that refers to the variable. For example, the expression `m = *p + 2` adds 2 to the value of `k` (the referent of `p`) and stores the result in `m`. Figure 16.2 shows two more examples of this indirect referencing.

<sup>2</sup>This nonuniform syntax causes confusion for many people. It permits an efficient implementation with an economy of notation at the expense of clarity. When C was a new language, efficiency was a major concern, and the C language developers expected only experts to use it.

An indirect reference accesses the value of the pointer's referent. These declarations create two `double` variables and two objects of class `PointT`. The assignments change the values marked by a large X.

```
double w = 16.2;
double* wp = &w;
double x;
PointT corner = {1.5, -2.0};
PointT* cp = &corner;
PointT c2;

x = *wp;    // Refers to a double.
c2 = *cp;   // Refers to an object.
w = cp->x; // Refers to a member.
```



**Figure 16.2. Indirect reference.**

The arrow operator, `->`, is used only for pointers that refer to objects. It gives a convenient way to dereference the pointer and select a member of the object in one operation. Therefore, in Figure 16.2, the expression `cp->x` would mean the same thing as `(*cp).x`; namely, “dereference the pointer `cp` and select the member named `x` from the object that is the referent of `cp`.” Programmers should use the “`cp ->`” notation rather than the clumsier “`(*cp)`” notation. (When the expression is written with the asterisk, parentheses are necessary because of the higher precedence of the `.` operator.) Pointers often are used to process arrays of structures; in such programs, the arrow operator is particularly convenient.

**Direct reference.** Like any other variable, we can simply refer to the value of a pointer, getting an address. This address can be passed as an argument to a function, stored in another variable, and so forth. For example, the assignment `p1 = p2`; copies the address from `p2` into `p1` so that both point at the same referent.

**Direct and indirect assignment.** C++ permits assignment between any two simple variables, objects, or pointers that have the same type. The meaning is the same for any type: The value of the expression on the right is copied into the storage area for the object on the left. With pointers, an assignment can be either direct, as in `p2 = ??`, or indirect, as in `*p2 = ??`. A direct assignment changes the value stored in the pointer and makes the pointer refer to a different variable. In contrast, an **indirect assignment**, which is written with an asterisk in front of the pointer name, changes the value of the underlying variable. For example, if `p` points at `k`, then `*p = n` means the same thing as `k = n`. Figure 16.3 gives an example of each of these assignments.

**Input and output.** Finally, let us look at the use of pointers with `>>` and `<<`. This is illustrated in Figure 16.4, where the pointer `ap` refers to `a[0]`. We use `ap` to read data indirectly into `a[0]`, then we read input directly into `a[1]`. Similarly, we print the first input indirectly, the second one directly.

**Printing a pointer.** The memory address contained in a pointer also can be printed and will appear in hex: `: cout << ap`. This technique can be useful when debugging a pointer program.

---

Indirect assignment changes the value stored in the pointer's referent; direct assignment changes the pointer itself.

Initial state:	After assignments:				
<pre>int k; int* p;  p = &amp;k;    // Direct assignment. *p = 17;  // Indirect assignment.</pre>	<table border="0"> <tr> <td style="text-align: center;"> <b>Initial state:</b>  <code>k</code>  <code>??</code>    <code>??</code> </td> <td style="text-align: center;"> <b>After assignments:</b>  <code>k</code>  <code>17</code> </td> </tr> <tr> <td style="text-align: center;"> </td> <td style="text-align: center;"> </td> </tr> </table>	<b>Initial state:</b> <code>k</code> <code>??</code> <code>??</code>	<b>After assignments:</b> <code>k</code> <code>17</code>		
<b>Initial state:</b> <code>k</code> <code>??</code> <code>??</code>	<b>After assignments:</b> <code>k</code> <code>17</code>				

**Figure 16.3. Direct and indirect assignment.**

Two declarations are given on the left, resulting in the memory layout shown. On the right are calls on `cin >> ap` and `cout << ap` and their results.

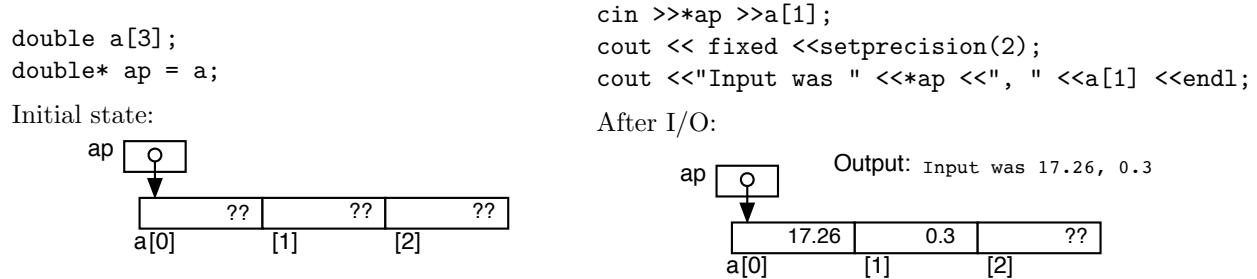


Figure 16.4. Input and output through pointers.

### 16.2.3 Pointer Arithmetic and Logic

Some, but not all, of the operations defined for numbers also are defined for pointers. Addition, subtraction, comparison, increment, and decrement are defined because these have reasonable and useful interpretations with pointers. Division, multiplication, modulo, and the logical operators are not defined for pointers because they have no meaning.

**Addition and subtraction with pointers.** All **pointer arithmetic** relates in one way or another to the use of pointers to process arrays. If a pointer points at some element of an array, then adding 1 to it makes it refer to the next array slot, while subtracting 1 moves a pointer to the prior slot. This is true of all arrays, not just arrays of 1-byte characters. A pointer and the array it points into must have the same base type, and the size of that base type is factored in when pointer arithmetic is performed. In Figure 16.5, `ip1` points at the beginning of the array, which has memory address 100. (The actual content of the pointer variable is the address 100.) Similarly, `ip2` points at the fourth slot of the array, which has memory address 106.

Adding `n` to a pointer creates another pointer `n` slots further along the array. In Figure 16.5, we set `ip1` to point at the beginning of the array `ages`. So `ip1 + 1` refers to the second array slot, at memory address 102. Now we can address any slot in the array by adding an integer to `ip1` or `ages`. As long as the integer is not negative and is smaller than the length of the array, the result always will be a pointer to a valid array slot.<sup>3</sup> For example, `ages[5]` is the last slot in the array. So, `ip1+5` refers to the same location as `ages[5]`.<sup>4</sup> Warning: The result of a pointer addition is an address, not a data value. Do not use pointer arithmetic instead of a subscript for normal processing. Use it only to set pointers.

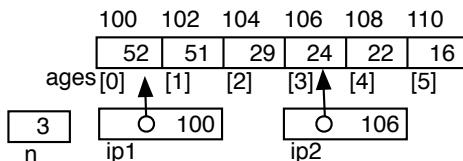
If two pointers point at slots in the same array, subtracting one from the other gives the number of array

<sup>3</sup>Any address resulting from pointer arithmetic refers to the beginning (not the middle) of an array slot because C adds or subtracts a multiple of the size of the base type of the array.

<sup>4</sup>Both also are synonyms for `ages+5`, because the name of an array is translated as a pointer to slot 0 of the array.

Pointers can be set to refer to an element in an array using the array name and the element's index value. In the diagram, subscripts are given below the array and memory addresses above it.

```
short int ages[6] = {52, 51, 29, 24, 22, 16};
int* ip1 = ages;
int* ip2 = ages + 3;
int n = ip2 - ip1;
```



If two pointers point at elements in the same array, subtracting one from the other tells you how many array slots are between them.

Figure 16.5. Pointer addition and subtraction.

A pointer can traverse the elements of an array using pointer arithmetic. In the diagram, subscripts are given below each array and memory addresses above it.

```
int* ip1 = ages;
ip1++;
```

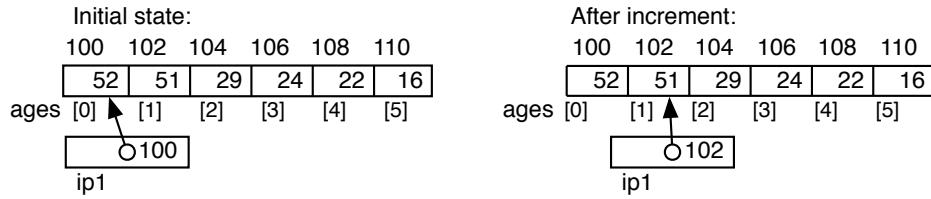


Figure 16.6. Incrementing a pointer.

slots between them. In Figure 16.5,  $ip2 - ip1$  is 3. The result of the subtraction  $ip2 - ip1$  is 3, not 6, because all pointer arithmetic is done in terms of array slots, not the underlying memory addresses.

Pointer subtraction is routinely used to find out how many data values have been input and stored in an array. If `head` is a pointer to slot 0 of the array and `last` is a pointer to the first unfilled array slot, then `last - head` is the number of values stored in the array.

**Increment and decrement with pointers.** Sometimes, pointers, rather than integer counters, are used to control loops. The increment and decrement operators make pointer loops easy and convenient. Increment moves a pointer to the next slot of an array; decrement moves it to the prior slot. For example, in Figure 16.6, the pointer `ip1` is initialized to the beginning of the array `ages`, then it is incremented. After the increment, it points to `ages[1]`.

**Pointer comparisons.** All six comparison operators are defined for use with two pointers that refer to the same data type. Two pointers are equal (`==`) if they point to the same slot, unequal otherwise. In Figure 16.7, `(ip1 == ages)` is true but `(ip1 == ip2)` is false. In general, a pointer `ip1` is less than a pointer `ip2` if the referent of `ip1` is a slot with a lower subscript than that of the referent of `ip2`. Pointer arithmetic can be used in these **pointer comparisons**. In the diagram, `(ip1 < ip2)` is true, as is `(ip2 < ip1+5)`.

#### 16.2.4 Using Pointers with Arrays

There is a very close relationship between pointers and arrays: A pointer to an array can be used with a subscript, just like the array itself. The difference between a pointer to an array and the name of an array is that the latter refers to a particular area of storage, while the former acts like a pronoun that can refer to any array or any part of an array. In this section, we show how pointers can be used to process an entire array.

Comparing pointers or addresses is just like comparing integers.

```
short int ages[6] = {52,51,29,24,22,16};
short int* ip1 = ages;
short int* ip2 = ages + 3;
short int* ipend = ages + 6;
if (ip1 == ip2) ... // false
if (ip1 > ip2) ... // false
if (ip2 < ipend) ... // true
```

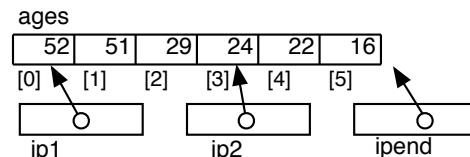


Figure 16.7. Pointer comparisons.

Suppose we start with the declarations on the left. Array `ages` contains six integers. A cursor is initialized to the head of `ages` and an off-board sentinel is set to point at its end.

```
#define N 6
short int ages[N] = {52, 51, 29, 24, 22, 16};
short int* cursor = ages;
short int* offBoard = ages + N;
```

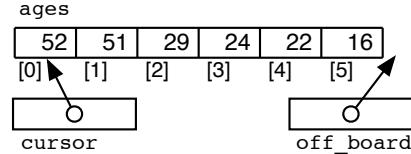


Figure 16.8. An array with a cursor and a sentinel.

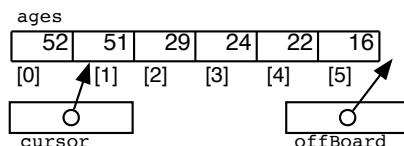
A major use of pointers in C++ is sequential processing of arrays, especially dynamically allocated arrays. Three kinds of pointers are commonly used for this purpose: head pointers, scanning pointers, and tail pointers. A **head pointer** stores the address of the beginning of the array that is returned by `new` when an array is allocated dynamically. The name of an array that is declared with dimensions can be treated like a head pointer in most contexts. A **tail pointer** is initialized to the end of an array. Both head and tail pointers, after they are set, should remain constant during their useful lifetime, and the head pointer is used to free the storage. In contrast, a **scanning pointer**, or **cursor**, usually starts at the head of an array, points at each array slot in turn, and finishes at the end of the array.

We can use pointer arithmetic to cycle through the elements of an array with great efficiency and simplicity. The scanning pointer initially is set to the head of the array, as shown in Figure 16.8. The tail pointer, often called a *sentinel*, is used in the test that terminates the loop. It points either at the last slot in the array (**on-board sentinel**) or at the first location past the end of the array (**off-board sentinel**). During scanning, all unprocessed array elements lie between the cursor and the sentinel. To process the entire array, we use a **scanning loop** to increment the `cursor` value so that it scans across the array, dereferencing it (`*cursor`) to access each array element in turn. We leave the loop when the cursor reaches the end of the array; that is, when it surpasses an on-board sentinel or bumps into an off-board sentinel. Figure 16.9 shows such a loop that will print the contents of an array. The upper diagram shows the positions of the pointers after the first pass through the loop. The lower diagram shows the positions of the pointers after exiting from the loop.

Starting with the declarations in Figure 16.8, we write a loop to process the `ages` array:

```
for (cursor = ages; cursor < off_board; ++cursor) {
    cout << *cursor ;
}
```

After the first pass through the loop, the output is: 52



After exiting from the loop, the output is: 52 51 29 24 22 16

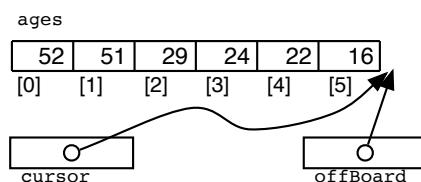


Figure 16.9. An increment loop.

**Measuring progress with pointer subtraction.** We can use pointer arithmetic to calculate how much of an array has been processed and how much remains. Remember that, if `pp1` and `pp2` are pointers and both are pointing at the same array, then `pp2 - pp1` tells us how many array slots lie between `pp1` and `pp2` (including one end of the range). Remember also that an unsubscripted array name is translated as a pointer to the beginning of the array. Therefore, if we are processing an array sequentially, we can calculate how many array slots already have been processed by subtracting the array name from a cursor pointing to the first unprocessed slot. For example, consider the loop in Figure 16.9. At the beginning of the loop, `cursor-ages == 0`; after the first pass through the loop, `cursor-ages == 1` because the cursor has been incremented and one data value has been printed. Similarly, subtracting the cursor from a sentinel tells us how many array slots are left to process between the cursor and the end of the array. For example, after the increment operation in Figure 16.9, `offBoard - cursor` is 5. This information can be useful in any program that processes an array using pointers.

**Subscripts vs. pointers.** Programmers converting to C or C++ from other languages often prefer to use subscripts. However, experienced C programmers use pointers more often, because once mastered, they are simple and convenient to use. Many applications of arrays use the array elements sequentially, visiting the slots in either ascending or descending order. For these situations, pointers provide a more concise and more efficient way to code the application. Every time a subscript is used, the corresponding memory address must be computed. To do so, the compiler generates code to multiply the subscript by the size of the base type, then adds the result to the base address of the array. In contrast, when you increment a pointer, the compiler generates code to add the size of the base type to the contents of the pointer. No multiplication is needed, just one addition. Also, once a pointer has been set to refer to a desired location, its value can be used many times without further computations. So, overall, sequential processing is more efficient with pointers than with subscripts. Therefore, experienced programmers often prefer using pointers when an array is to be processed sequentially but use subscripts for nonsequential access to array elements or for processing parallel arrays.

## 16.3 Dynamic Memory Allocation

When an array is declared with a constant length, like the `dataList` array in Figure 10.29, a C or C++ translator calculates the size of the array at compile time. At run time, the predetermined amount of storage is allocated and the program must work within fixed array boundaries. In many applications, though, the amount of data to be processed and, therefore, the length of an array to store the data are not known ahead of time. A sort program is a good example of an application that may operate on a small or large amount of data; the operation of the program is not tied to any particular amount of data. However, defining a maximum array length at compile time limits the usefulness of a sort program to data sets smaller than that maximum.

We can eliminate this artificial restriction by using pointers and **dynamic memory allocation**. We can write a program that can sort any amount of data that will fit into the memory of the computer. If the initial amount of memory is inadequate, the array is resized to be able to contain the entire data set.

This makes the program much more flexible than one with a #defined array length. Image processing and graphics applications profit from dynamically allocated memory, because images and graphical objects come in many sizes, from small to large, but the processing methods remain the same.

Both C and C++ support a set of functions for creating and handling dynamically allocated memory. When given the required size of a memory area, these functions interact with the operating system and ask it to reserve an additional block of memory for the program's use. The beginning address of this block is returned to the program as a pointer value that can be saved and later used to access the memory. The dynamic-memory functions for C++ are listed below and described in more detail in the subsections that follow. The corresponding functions in C are listed in Appendix E.

**Dynamic memory allocation functions.** These functions are defined in C++.

- `new TypeName;`  
Allocate a block of memory large enough to store this type and some bookkeeping information. Initialize the memory using the `TypeName` constructor. Return a pointer to the beginning of the initialized object<sup>5</sup>.

---

<sup>5</sup>New is analogous to malloc and calloc in C

This is the conceptual model of what happens when you call `new`.



**Figure 16.10. Allocating new memory.**

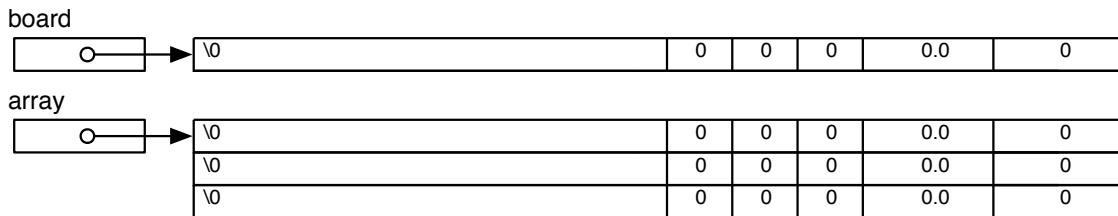
- `new TypeName[n];`  
Allocate a block of memory large enough to store an array of  $n$  objects of this type, plus some bookkeeping information. Initialize the memory by using the `TypeName` default constructor  $n$  times. Return a pointer to the beginning of the array. The array length,  $n$  must be stored as part of this allocation because it is needed later to free the memory.
- `delete pointerName;`  
Recycle the memory block pointed at by the `pointerName`<sup>6</sup>. Return it to the operating system for possible future use. A block that was created using `new` should be deleted when it no longer is needed by the program. Objects that were created by declarations must never be manually deleted. (Deletion for local variables happens automatically when the variable goes out of scope.)
- `delete[] pointerName;`  
Recycle the entire memory block pointed at by the `pointerName`. First, however, run the destructor for the base type of the array  $n$  times to free all memory attached to the objects stored in the array.

### 16.3.1 Simple Memory Allocation

When a programmer cannot predict the amount of memory that will be needed by a program, the `new` can be used at run time to allocate the desired number of bytes. For example, we can use `LumberT` from Chapter 13:

```
LumberT* board = new LumberT;           // Allocate one object.
LumberT* array = new LumberT [3];        // Allocate three objects.
```

Before returning, the default constructor for `LumberT` will be called to initialize the objects. Conceptually, this code allocates memory areas like the ones in Figure 16.10



Actually, the allocation is somewhat more complex, although the programmer rarely needs to know about it, and the implementation is not covered by the standard. The truth is, some additional storage is allocated to allow the system to manage the dynamic memory. The area is, at least, the size of a long integer, and the location is normally immediately preceding the first byte of the new object. The gray area in the diagram represents these additional bytes that the C++ system sets aside; it stores the total size of the allocated block (the size of a `size_t` value plus the size of the white area). The importance of these bytes becomes clear when we discuss `delete`.

In old C, it was necessary to check for the success of any request for dynamic allocation. In C++, that is not necessary and not appropriate. If the computer does not have enough memory to satisfy the request for a new allocation, a `bad_alloc` exception will be thrown. Unless an exception handler is written as part of the code, this will terminate execution.

<sup>6</sup>Delete is analogous to free in C

### Freeing Dynamic Memory

In many applications, memory requirements grow and shrink repeatedly during execution. A program may request several chunks of memory to accommodate the data during one phase then, after using the memory, have no future need for it. Memory use and, sometimes, execution speed are made more efficient by recycling memory; that is, returning it to the system to be reused for some other purpose. Dynamically allocated memory can be recycled by calling `delete`, which frees a block of memory by returning it to the system's memory manager. The number of bytes in the gray area at the beginning of each allocated block determines how much memory is freed. The use of `delete` is illustrated in the `Flex` class in Figure 16.11.

While each program is responsible for recycling its own obsolete memory blocks, a few warnings are in order. A block should be deleted only once; a second attempt to delete the same block is an error. Similarly, we use `delete` only to recycle memory areas created by `new`. Its use with a pointer to any other memory area is an error. Another common mistake, described next, is to attempt to use a block after it has been deleted. These are serious errors that cannot be detected by the compiler and may cause a variety of unpredictable results at run time.

### Dangling Pointers

A **dangling pointer** is one whose referent has been reclaimed by the system. Any attempt to use a dangling pointer is wrong. Typically, this happens because multiple pointers often point at the same memory block. When a block is first allocated, only one pointer points to it. However, that pointer might be copied several times as it is passed into and out of functions and stored in data structures. If one copy of the pointer is used to free the memory block, all other copies of that pointer become dangling references. A dangling reference may seem to work at first, until the block is reallocated for another purpose. After that, two parts of the program will simultaneously try to use the same storage and the contents of that location become unpredictable.

To avoid the problems caused by dangling pointers, the programmer must have a clear idea about which class, and which variable within the class, is responsible for keeping the “master pointer” to each dynamic object, and later, for freeing the dynamic memory.

### Memory Leaks

If you do not explicitly free the dynamically allocated memory, it will be returned to the system's memory manager when the program completes. So, forgetting to perform a `delete` operation is not as damaging as freeing the same block twice.

However, some programs are intended to run for hours or days at a time without being restarted. In such programs, it is much more important to free all dynamic memory when its useful lifetime is over. The term *memory leak* is used to describe memory that should have been recycled but was not. Memory leaks in major commercial software systems are common. The symptoms are a gradual slowdown in system performance and, eventually, a system “lock up” or crash.

Thus, it is important for programmers to learn how to free memory that is no longer needed, and it is always good programming style to do so. This is even true when the memory is used for a short time by only one function. Functions often are reused in a new context, they always should clean up after themselves.

### Using Dynamic Arrays

A pointer to a dynamic array can be used with a subscript, just like the array itself, as shown in this code fragment below that allocates space for `n` long integers, then reads that many numbers from `cin`:

```
lptr = new long[n];
for (int k = 0; k < n; ++k) cin >> lptr[k];
```

The close relationship between an array pointer and an unsubscripted array name makes it very easy to take an application written for ordinary arrays and convert it to use dynamic arrays. As an example, consider the selection sort program from Figure 10.29. This program consists of `main()` and four other functions and uses an array whose length is defined at compile time. To allow this program to use dynamic allocation in C++, only six lines in the main program need be changed; namely, the lines that determine the length of the array and allocate it. None of the function definitions or prototypes need to be modified. We will revisit the selection sort program, written in C++ later in this chapter after presenting dynamic arrays that can grow, as needed.

## 16.4 Arrays that Grow

When we declare an array length in a program, we then must write code to guard that array and ensure that no part of the program walks off the end of the array and stores information into adjacent memory cells. Array-based programs become longer and more complex because of these efforts. The situation is worst when allocating an array to hold input. How do we guess how much input there will be? How do we detect when there is not enough space, and what do we do if there is more input?

Using `new` to allocate arrays at run time is the first step in removing restrictions on the length of an array. However, that array space still must be allocated before the data are read and before we know how many data items actually exist. A dynamic array still cannot accommodate an amount of data greater than expected.

**Flex Arrays.** To remove the size restrictions, we need to be able to *resize the data array*. We want to replace the existing (too small) memory block by a larger one that contains the same data. To do this we define a class that contains the dynamic array, the two integers needed to manage it, and functions to handle resizing and related needs. The `Flex` class needs three data members:

- The dynamic array.
- The current capacity of the array, *max* (the number of slots in it).
- The number of data items currently stored in the array, *n*

At least two function members are needed. The first is `push_back()`, which puts data into the first unoccupied slot in array. The second is `grow()`, which actually does the resizing. The `push_back()` method has two steps:

- Check whether there is currently space in the array for another data item. If not, call `grow()`.
- Whether or not growth happened, put the additional data item into the first empty slot in the array, and increment *n*.

The `grow()` method has several steps:

- Use a temporary variable to point at the existing array.
- Double *max*. By doubling the array capacity each time, we guarantee that the time required to reallocate and copy data will always be acceptable, and that the total number of data items copied in the lifetime of the array will not exceed the current length of the array.
- Allocate a new memory area with the new *max* capacity.
- Copy *n* data objects from the old memory into the new area.
- Delete the old memory area.

Finally, C++ allows a program to define the subscript operator for that class. Doing so allows the programmer to treat the `Flex` array as if it were an ordinary array.

### 16.4.1 The Flex Class

The C++ language supports a sophisticated and generic growing array type named `vector`. A practicing programmer would use `vector` when storage needs are unpredictable. This section presents a much simpler class named `Flex` that works the same way as `vector` but is easier to study. By learning how `Flex` works, the student learns how `vector` works.

**Notes on Figures 16.11 and 16.12: Flex: a growing array.** This is the header file for the `Flex` class. The function definitions are in Figure 16.12. This class is used in the selection sort program in Figure 16.19.

#### *First box: dealing with the environment.*

- This is the first example program that is composed of three classes and a main program, and the first where a module needs to `#include` two user-defined header files. At this level of complexity, we begin to have a problem with the `#include` statements: the code will not compile with too few of them, or if a header is included twice in the same module or if there is a *circular include*.

```

#pragma once
#include <iostream>
using namespace std;
#include "trans.hpp"
typedef Transaction BT;

#define START 4           // Default length for initial array.

class Flex {
private: // -----
    int max = START;      // Current allocation size.
    int n = 0;            // Number of array slots that contain data.
    BT* data = new BT[max]; // Allocate dynamic array of base type BT.

    void grow();          // The Flex is full, so double the allocation.

public: // -----
    Flex() {}            // An array of Transactions.
    ~Flex() { delete[] data; } // Free dynamically allocated data.

    void push_back( BT data ); // Store data and return the subscript.
    int size() { return n; }   // Provide read-only access.
    BT& operator[]( int k );
    void print(ostream& out) { for (int k=0; k<n; ++k) out << data[k] << endl; }

};


```

**Figure 16.11. Flex: a growing array.**

- The `#pragma` statement on the first line tells the compiler to include this file once, but if it has already been included in a module, do not include it again. It is good style to include the `#pragma once` whether or not it is needed. In this particular program, it is needed in the file `trans.hpp` but not in the files `charges.hpp` and `flex.hpp`.

**Second box: dealing with the environment.**

- This is a generic class, defined in terms of an abstract base type, BT. This class can be used to hold data of any base type by defining BT to mean the target type. We use Flex in the selection sort program to hold a series of transactions. To make that possible, we need to include the transaction header here.
- We use a `typedef` to say that the BT, is really the Transaction class.

**Third and fourth boxes: Private data members.**

- In Flex, as in most classes that define data structures, the data members of the class must not be modified by any other class. If they *were* modified, the data structure would malfunction. Thus, the data members are private.
- In this class, it is possible to give meaningful initializations for the data members at compile time. We choose an arbitrary small number, `START` as an initial allocation size. If it is too large, little memory is wasted. If it is too small, it will grow. We don't need to guess, and this number does not need to depend on the application.
- Since the initial array length is a constant, and a new Flex is always empty, it is possible to initialize all of the data members in the class declaration. That permits us to have a constructor that does nothing.

```

#include "flex.hpp"
// ----- Store object in array. Grow first, if needed.

void Flex :: push_back( BT obj ) {
    if ( n == max ) grow();           // Create more space if necessary.
    data[n] = obj;
    ++n;
}

// ----- Double the allocation length.

void Flex :: grow() {
    BT* temp = data;                // Hang onto old data array.
    max *= 2;
    data = new BT[max];             // Allocate a bigger one.
    for (int k=0; k<n; ++k) data[k] = temp[k]; // Copy info into new array.
    delete temp;                   // Recycle (free) old array.
}

// ----- Access the kth char in the array.

BT& Flex :: operator[]( int k ) {
    if ( k >= n || k < 0 ) fatal("Flex bounds error.");
    return data[k];                // Return reference to desired array slot.
};

```

Figure 16.12. Flex functions.

*Fifth box: a private function.*

- Just as it would be disastrous to let another class modify *n* or *max*, it would make a mess if another class called *grow()* at the wrong time. Therefore the *grow()* function is private.
- In this class, it is possible to give meaningful initializations for the data members at compile time. We choose an arbitrary small number, *START* as an initial allocation size. If it is too large, little memory is wasted. If it is too small, it will grow. We don't need to guess, and this number does not need to depend on the application.
- Since the initial array length is a constant, and a new Flex is always empty, it is possible to initialize all of the data members in the class declaration. That permits us to have a constructor that does nothing.

*Sixth box: the constructor and destructor.*

- The Flex constructor is empty; there is no work to be done. All the initializations were done in the fourth box.
- The Flex class does dynamic allocation, therefore it must take the responsibility of freeing the dynamic memory when it is no longer needed. This is what a destructor is for.
- The name of a destructor is a tilde (~) followed by the name of the class. All dynamic memory blocks created anywhere in the class must be deleted by the destructor.
- There is only one dynamic memory block, an array, and its pointer is stored in *data*. Therefore, we write *delete[] data;*

*Seventh box and Figure 16.12: the class functions.*

- Two of the class functions are inline, and one-line definitions are given here. They are `size()` and `print()`. The other three functions, `grow()`, `push_back()`, and `operator[]` are defined in `flex.cpp`, the implementation file, which is shown in Figure 16.12 and included in the comments here.
- The names `push_back()` and `size` are the same as the names of the corresponding function in the standard `vector` class.
- **`push_back()`.**  
This function is the only correct way to store new data in the Flex array. It tests for a full array and, if found, extends the array. The class member `n` is always an accurate count of the number of objects in the array.
- **`grow()`.**  
The `grow()` function was explained in Section 16.4. It is a private function because only the Flex class should ever use it. It is called when there is a request to store another thing in the Flex array but the array is full.
- **`subscript()`.**  
To define the subscript operator for this class, we write a definition of `operator []( int )`; Writing operator definitions is beyond the scope of this text. However, an operator definition is necessary here to allow subscript to be used in the ordinary way. The student does not need to understand this code at this time.
- In this definition, the Flex class delegates the subscript operation to the dynamic array and returns the value that the array subscript returns. The return value is the address of one slot in the array. This allows us to use subscript for either reading or writing an array element.
- Note that the code performs a bounds check, and will not allow storing into a subscript outside of the filled portion of the Flex array.

## 16.5 Application: Insertion Sort

The first application of the Flex array is an insertion sort algorithm. Insertion sort is one of two easy sorting algorithms that are often useful and should be learned<sup>7</sup>.

On the surface, the insertion sort algorithm looks a lot like the selection sort studied earlier; both have an outer loop that executes  $n - 1$  times to sort  $n$  items. Both have an inner loop, but this is where the similarity ends. When doing selection, we repeatedly examine the remaining *unsorted items* to find the largest one. We must look at all of them to find the largest. When found, we swap it with the last unsorted value in the array.

In the **insertion sort** algorithm, the data array is divided into two segments: the sorted portion first, followed by the unsorted portion. To sort, we pick the first unsorted item and then scan backwards through the list of *sorted items* from the high end to the beginning of the array, looking for the correct position for the new item. As soon as we locate an item smaller than the one being inserted (assuming we sort in ascending order), we stop looking. On the average, this search for the correct insertion slot takes fewer comparisons than looking through the remaining unsorted items to find the next smallest item during a selection sort.

Since we can't just create new space for a value between two others in an array; we have to move some of them to make space for an insertion. (See Figure 16.13.) We do this by shifting each sorted data item one slot toward the end of the array as we pass it during our search, in essence moving a hole backwards along the array, so that it is always adjacent to the item we are testing. When the proper insertion slot is found, we simply store the current value in our hole. This sequence of movements requires more work than the single data swap in a selection sort. However, because the number of comparisons needed is smaller, insertion sort usually is faster.

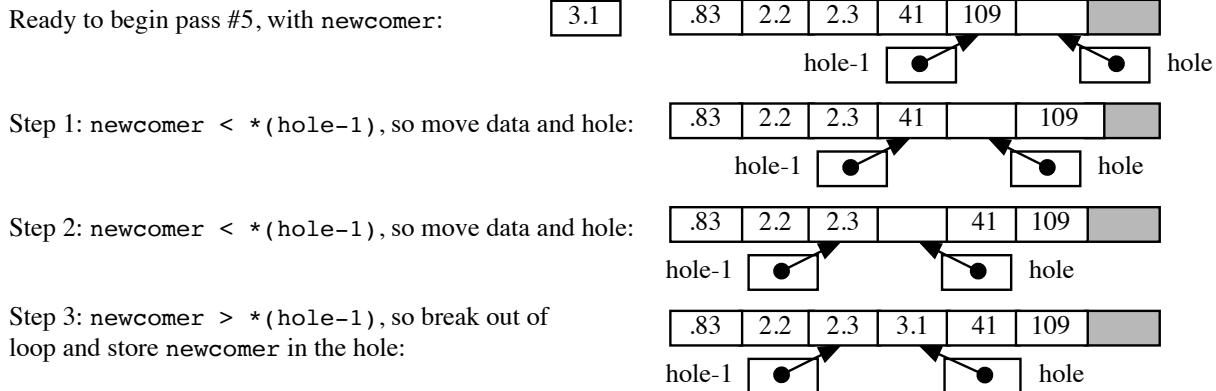
This insertion process is repeated  $n-1$  times. After each of the  $n-1$  passes, the sorted part is one item longer and the unsorted part is one item shorter. An example is illustrated by the diagrams in Figure 16.16, which show the intermediate states of an array of 10 items after each pass.

The insertion algorithm is implemented by the function `insertionSort()` listed in Figure 16.18. Pointers are used, instead of subscripts, to access the data array. The main program for insertion sort is in Figure 16.15.

---

<sup>7</sup>The other useful easy sort is selection sort, which is presented later in this chapter. A third sort in the same general category is bubble sort, which should never be used for any purpose because it can take twice as long to execute as insertion sort.

This represents the fifth pass through the array.



**Figure 16.13. Inner loop of the Insertion sort.**

Following that are the controller class, **Sorter** and the data class, **Transaction**. The structure of the application is diagrammed in Figure 16.16.

The controller class uses a Flex array to store all the data from an input file. Flex arrays are dynamically allocated data structures that create, manage, and free the dynamic memory they use. The client class, in this case **Charges**, can rely on this dynamic flexibility but does not need to take any responsibility for it, since all management is handled by the Flex class.

**Notes on Figure 16.15: Insertion sort.** This is the main program for the insertion sort. It uses the **Sorter** class defined in Figures 16.17 and 16.18 and the Flex array class in Figures 16.11 and 16.12.

#### *First box: Dealing with the environment.*

- This main program uses the class **Sorter**, and includes the header for that class.
- The **#define** statement supplies the name of the input file as a quoted string, a form that is appropriate for opening the file.

#### *Second box: Opening the file.*

- The input stream is declared and opened in the same line. This is better style than using two lines to do the two actions separately.
- As usual, we test for a properly opened file. The error comment has three parts: it begins and ends with a quoted string and has the **#defined** symbol between the quoted strings. This uses two advanced features of the C language: macros and string merging.
- Macros: **#define** commands are interpreted at compile time, not at run time. Wherever you write the **#defined** symbol, the compiler's preprocessor removes what you wrote and replaces it by the definition of the symbol. So these two lines:

```
ifstream fIn( FILE );
if (!fIn.good()) fatal( "Cannot open " FILE " for reading." );
```

are replaced by this code prior to compilation:

```
ifstream fIn( "insr.in" );
if (!fIn.good()) fatal( "Cannot open " "insr.in" " for reading." );
```

- String merging: Whenever two strings are written in a program and the quote marks are separated by nothing except possible whitespace, the compiler will merge the two strings into a single string. In this case, the result from the preprocessor is three adjacent strings, and they are merged to form one string, thus:

```
if (!fIn.good()) fatal( "Cannot open insr.in for reading." );
```

This is what the compiler sees and compiles: a single string.

Sorting 10 items with a pointer-based insertion sort.

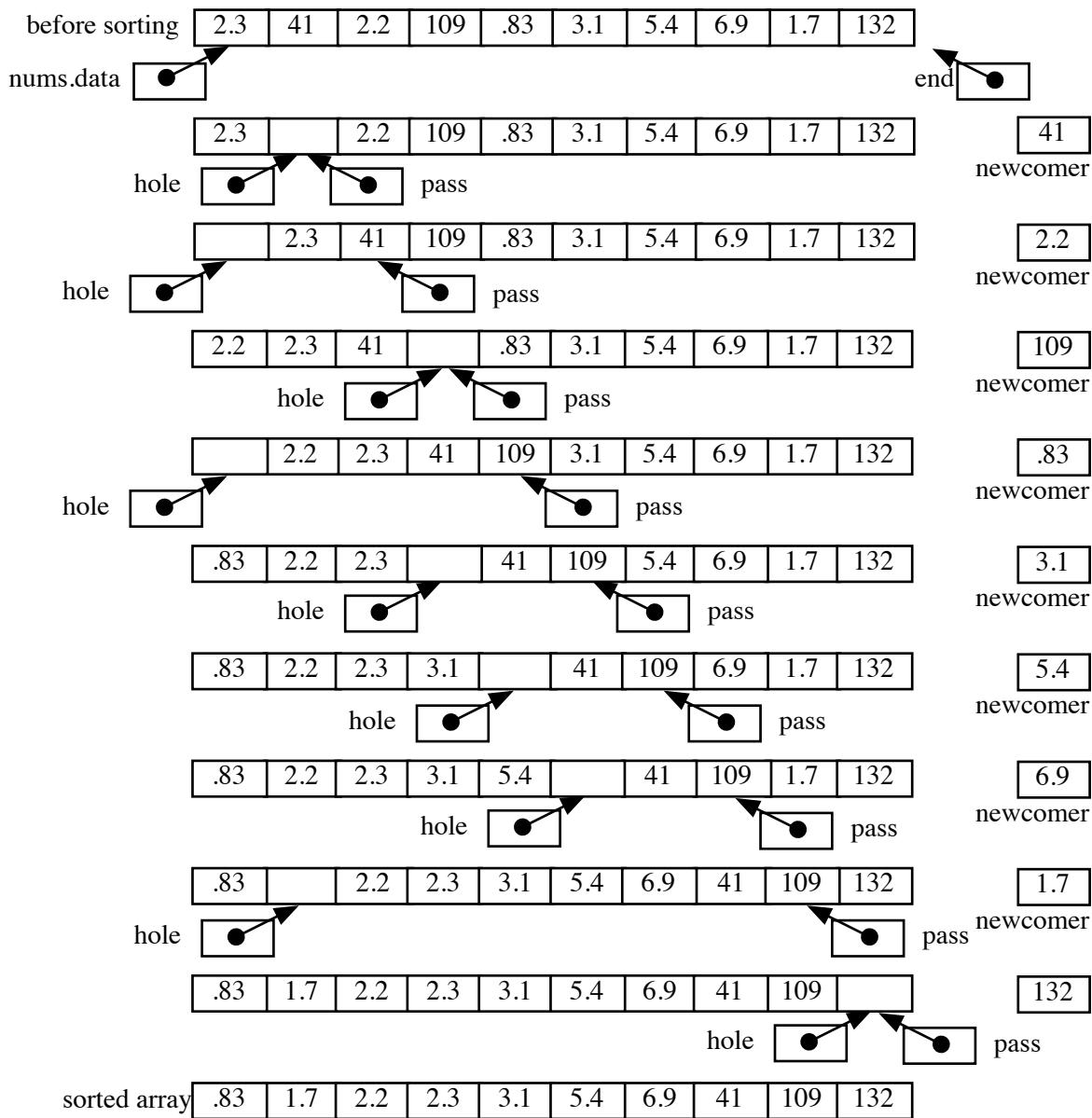


Figure 16.14. Insertion sort.

```
#include "sorter.hpp" // File: main.cpp
#define FILE "insr.in"

int main( void ) {
    cout << "Insertion Sort Program for type float\n";
    ifstream fIn( FILE );
    if (!fIn.good()) fatal( "Cannot open " FILE " for reading." );

    Sorter nums( fIn );
    cout << nums.getN() << " numbers were read; beginning to sort.\n";
    nums.insertionSort();
    cout << "\Data sorted, ready for output\n";
    nums.print( cout );
    return 0;
}
```

Figure 16.15. Insertion sort using a dynamic array.

**Third box: Doing the work.**

- This main function follows the ordinary form for an OO main program: (1) Call the constructor to create an object. (2) Use it to get the work done. (3) Print the results.
- We don't try to sort the data or print it here in the main function. The Sorter class is the expert on this data set, so we delegate the sorting and printing operations to the expert.
- After each phase is a line of output. This kind of output is important during program development and debugging. It also lets the user know what is going on inside the program.

**Notes on Figures 16.17 and 16.18: The Sorter class.** This is the header file for the Sorter class. The function definitions are in Figure 16.18. This class is used in the insertion sort program in Figure 16.15.

A call-chart for the insertion sort.

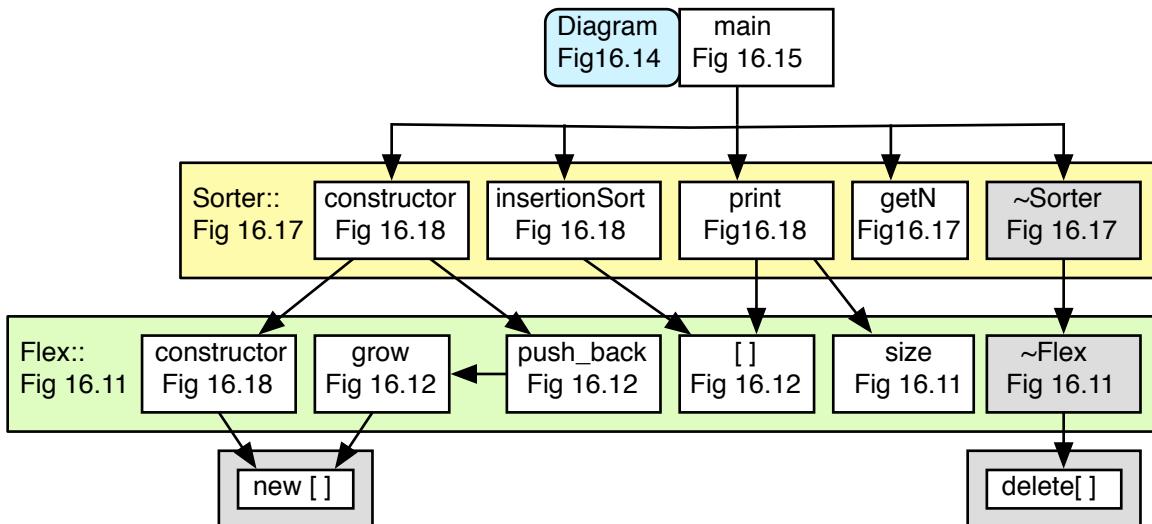


Figure 16.16. Insertion sort.

```

#pragma once                                // File: sorter.hpp
#include "tools.hpp"
#include "flex.hpp"

class Sorter {
private:
    Flex data;                                // Dynamic data array.
    istream& inFile;

public:
    Sorter( istream& inF );                  // Constructor for the controller.

    void print( ostream& out );              // Output the data array.
    void insertionSort();
    int getN() { return data.size(); }
};


```

Figure 16.17. The Sorter class.

**First box: Dealing with the environment.**

This controller class instantiates a Flex array and uses it to store floats. Therefore, we include the header file for Flex arrays. By including the tools header file, we bring in all the standard headers we are likely to need, and the required namespace declaration.

**Second box: Data members.** There are only two data members: an open `istream&` to supply the data and a Flex array to store it. They are not initialized because we do not have the information, at compile time, to write meaningful initializations. Both will be initialized by the constructor.

**Third box Figure 16.17 and first box, Figure 16.18: the constructor**

- Main will open the input file and pass an open `istream&` as a parameter. The only way to use an `&` parameter is a ctor (constructor initializer), shown in the small box on the first line of the constructor. The ctor takes the parameter (named `inF`) and stores it in the data member named `inFile`.
- Bringing a database from a file into the program is something that a constructor often does. It is part of making the class ready to use.
- The second box in the constructor is a loop that reads one line of the input file at a time and does a minimal check for read errors and end-of-file. If either happens, the loop ends and the program processes the data it already has.
- Any illegal character in the file will end the input and will not warn the user about the problem. This is not a good way to design an application! It is done here as a bad example of file handling. The selection sort program later in this chapter does a better job of error handling.
- If there *was* good data, then we push it onto the end of the Flex array. If the array is full, it will grow.

**Fourth box in Figure 16.17: an accessor.** The `getN()` function is a typical accessor. The main function calls it in order to provide high-quality user feedback. It is not used by the sorting algorithm. It is inline to increase efficiency.

**Fourth box in Figure 16.17 and second box in Figure 16.18: sorting.**

- This sort algorithm is written to use pointers, not subscripts. Note that pointers can be incremented and decremented, just like subscripts, and that we use an asterisk here instead of a subscript to access a data value.

- Note that this code uses helpful names, `newcomer` and `hole`, instead of using variables named `i` and `j`. Using meaningful names makes code far, far easier to understand.
- The logic of the `insertionSort()` algorithm is as follows:
  1. To sort  $n$  items, write an outer loop that will execute  $n - 1$  times.
  2. In the body of the loop, pick up a copy of the first unsorted data value and call it the *newcomer*. A list of one item is always sorted, so start the first pass with the value in `data[1]`. This leaves a hole in the array at subscript 1.
  3. Each time around the outer loop, walk backwards from the hole to the head of the array. At each step compare the *newcomer* to the value in the current array slot. If the *newcomer* is smaller, move the other value into the hole, then decrement the hole-pointer and repeat the inner loop.

```
Sorter::Sorter( istream& inFile ) : inFile( inFile ) {
    float temp;
    int k;
    for (k=0; ; k++) {
        inFile >> temp;
        if ( ! inFile.good() ) break;
        data.push_back( temp );
    }
}
```

```
void Sorter::
insertionSort() {
    float* head = &data[0]; // The beginning of the array.
    float* end = head + data.size(); // Off-board sentinel.
    float* pass; // Starting point for pass through array.
    float* hole; // Array slot temporarily containing no data.
    float newcomer; // Data value being inserted on this pass.

    for (pass = &data[1]; pass < end; ++pass) { // Outer loop; n-1 passes.
        newcomer = *pass; // Pick up next item,
        // ...and insert into the sorted portion at the head of the array.
        for (hole = pass; hole > head; --hole) {
            if (*(hole-1) <= newcomer) break; // Insertion slot is found.
            *hole = *(hole-1); // Move item back one slot.
        }
        hole = newcomer;
    }
}
```

```
void Sorter::
print( ostream& outFile ) {
    float* cursor = &data[0];
    float* end = cursor + data.size(); // an off-board sentinel
    for( ; cursor<end; ++cursor) { // Use a pointer to walk the array.
        outFile << *cursor << '\n';
    }
}
```

Figure 16.18. Insertion sort using a Flex array.

4. Eventually, we come to either the end of the array or a slot containing a value smaller than `newcomer`. At this point, the current `hole` position is where `newcomer` must be inserted, so we terminate the inner loop and put the *newcomer* in the hole.
5. We might shift as few as no items or as many as currently are in the sorted portion of the array. On the average, we move about half of the sorted items.
6. The data is sorted after the last value in the array has been inserted into its place.

***Fourth box in Figure 16.17 and third box in Figure 16.18: printing.***

- The print function uses a pointer-loop to access the data values. To set it up, pointers are defined for the beginning and end of the data in the Flex array.
- As is customary in C and C++, the end pointer is *off-board*, that is, it points at the first unoccupied array slot, not the last occupied slot. Using an off-board pointer makes it easier to write a loop. The language standard guarantees that this is OK.
- At each step, one number is printed with whitespace to separate it from all the other output. It is easy to forget that the whitespace is essential. When forgotten, the output is unreadable.

## 16.6 Selection Sort

In this section, we present a second version of the selection sort algorithm. This algorithm was first presented in C in Figure 10.29. The version presented here has been transformed to an object-oriented design implemented in C++. Although the logic is the same, the organization of the code is totally changed. The main program is now brief and all the work is done in a controller class (`Charges`) and a data class (`Transaction`).

The form and content of this program are very much like the insertion sort program, above. Details are different, and the complexity is greater because the data being sorted are objects, not simple numbers. The controller class uses a Flex array to store the data from an input file. The client class, in this case `Charges`, can rely on this dynamic flexibility but does not need to take any responsibility for it, since all management is handled by the Flex class.

**Notes on Figure 16.19: Selection sort.**

***First box: Dealing with the environment.***

- This main program uses the class `Charges`, and includes the header for that class.
- The `#define` statement supplies the name of the input file as a quoted string, a form that is appropriate for opening the file.

***Second box: the body of main().*** The code here is exactly parallel to the insertion sort code in Figure 16.15. The comments will not be repeated here.

**Notes on Figure 16.20: The Charges class.** This is the header file for the `Charges` class. The function definitions are in Figure 16.21. This class is used in the selection sort program in Figure 16.19. The unboxed parts of this class are exactly parallel to the `Sorter` class in Figures 16.17 and 16.18, so the comments will not be repeated here.

***First box: Dealing with the environment.***

This controller class instantiates a Flex array and uses it to store `Transactions`. Therefore, we include the header files for both classes.

***Second box in Figure 16.20 and second box in Figure 16.21: a private function.***

This class has one private function and three public functions. The `findMaxID()` function is private because it intended for use by the public sort function, and not for use by a client class. The code could be written as an inner loop in the sort function, but the selection sort algorithm is clearer this way.

---

This is the main program for the selection sort. It uses the controller class, Charges, in Figures 16.20 and 16.21. The data class, Transaction, is in Figures 16.22 and 16.23.

```
#include "charges.hpp"
#define FILE "charges.txt"

int main( void ) {
    cout << "Club Snack Bar Accounting Program\n";
    ifstream fIn( FILE );
    if (!fIn.good()) fatal( "Cannot open " FILE " for reading." );
    Charges snacks( fIn );
    cout << snacks.getN() << " transactions were read; beginning to sort.\n";
    snacks.sort();
    cout << "\nData sorted, ready for output\n";
    snacks.print( cout );
    return 0;
}
```

---

**Figure 16.19.** Selection sort using a dynamic array.

---

This is a controller class. It is instantiated by the main function in Figure 16.19 and carries out the logic of the application. To do this, it creates and operates on an array of data objects (Transactions).

```
#pragma once                                // File: charges.hpp
#include "trans.hpp"
#include "flex.hpp"

class Charges {
private:
    const char* name = "ECECS Snack Club";
    istream& inFile;
    Flex mems;
    int findMaxID( int last );

public:
    Charges( istream& in );
    int getN() { return mems.size(); }
    void sort();
    void print( ostream& out );
};
```

---

**Figure 16.20.** The controller class: Charges.

**Third box *Figure 16.20* and first box, *Figure 16.21: the constructor***

- The first inner box in the constructor reads one line of the input file and does a complete error check. The first if statement looks for a normal end of file and leaves the input loop if it is found.
- The second if statement tests for all other input errors. If the stream state is not good, no further input can happen. It must be corrected. We reset the stream's error flags to the good state by calling `clear()`.
- The most minimal action that could possibly correct the problem is to eliminate the character that caused the input error. `ignore(1)` does that job.
- If there was good data, then we use it to instantiate a new transaction and immediately put the transaction into the Flex array. If the array is full, it will grow.

**Fourth box in *Figure 16.20* and third box in *Figure 16.21: sorting.***

- The `sort()` function implements a selection sort:
  1. To sort  $n$  items, write an outer loop that will execute  $n - 1$  times.
  2. Each time around the loop, find the largest value in the array between slot 0 and slot  $n - 1$ , and swap it to the end of the array.
  3. Subtract one from  $n$  and repeat the loop.
- Note that this code uses helpful names, `last` and `where`, instead of using variables named `i` and `j`. Using meaningful names makes code far easier to understand.

**Fourth box in *Figure 16.20* and fourth box in *Figure 16.21: printing.***

- The last class function, `print()`, relies on two major OO techniques: expertise and delegation. The Flex array is the expert on storing Transactions and it knows how many have been stored. Similarly, the Transaction class is the expert on formatting and printing a transaction. The Charges class is not expert on either of these things.

So `Charges::print()` calls on `mems`, the Flex array to find out how many items to execute its print loop, then calls `Transaction::print()` to do the printing. In both cases, the action is *delegated* to the expert.

**Notes on Figures 16.22 and 16.23: Transactions.** This is the data class. It is used in the selection sort program in Figure 16.19.

**First box, *Figure 16.22: Dealing with the environment.***

- This header file must be included twice: in `flex.hpp` and in `charges.hpp` because both classes refer to `Transactions`. Therefore the `#pragma once` declaration is essential.
- By including the tools header file here, we make it unnecessary to include it in `flex.hpp` and `charges.hpp`

**Second box, *Figure 16.22: Data members.***

This very simple class records one transaction made by one person. It has only a member's ID and the price of the item he took from the snack bar.

**Third box, *Figure 16.22* and second box in *Figure 16.23: Constructors.***

- There are two constructors in this class. The default constructor is called by the system when a `Charges` object is created because `Charges` contains a Flex array of `Transactions`. When an array is created for any type, a default constructor is required for that type.
- The second constructor is the most ordinary form there is: it has one parameter per data member, used to initialize the new object.
- The definition of this method in Figure 16.23 is also very ordinary: one assignment statement per pair of a parameter and a data member. The parameters have the same names as the data members of the class, so `this->` must be used to refer to the data members. If the names are different, `this->` is not needed.
- No destructor is needed because there is no call on `new` anywhere in this class.

```

Charges::Charges( istream& in ) : inFile( in ) {
    int ID;
    float owes;
    for (;;) { // Growing array will hold all data in file.
        inFile >> ID >> owes;
        if (inFile.eof()) return;
        if (!inFile.good()) { // Remove faulty line after input error.
            inFile.clear();
            cerr <<"Bad data on line " <<mems.size() <<endl;
        }
        mems.push_back( Transaction( ID, owes ) );
    }
}

int Charges::findMaxID( int last ) { // Find the maximum ID in the array
    int finger = 0; // Put your finger on the first ID.
    // Now compare the fingered value to the values that follow it.
    for (int cursor = 1; cursor < last; ++cursor) {
        // If next is bigger, move your finger to it.
        if (mems[cursor].bigger( mems[finger] )) finger = cursor;
    }
    return finger; // Your finger is on the biggest value.
}

void Charges::sort() {
    int last = mems.size(); // Number of actual data items in the array.
    int where; // Position of largest value in the index array.
    while (last > 0) {
        where = findMaxID( last-- );
        // Swap the two transactions.
        Transaction temp = mems[where];
        mems[where] = mems[last];
        mems[last] = temp;
    }
}

void Charges::print( ostream& out ){
    for( int k=0; k<mems.size(); ++k ) mems[k].print(out);
}

```

Figure 16.21. Functions for the Charges class.

---

This is a data class. It is used by the Charges class, and indirectly, by the main program in Figure 16.19.

```
#pragma once                                // File: trans.hpp
#include "tools.hpp"

// -----
class Transaction {
private:
    int ID;                      // Transaction number
    float owes;                  // Amount owed by Transaction

public:
    Transaction() = default;
    Transaction( int ID, float owes );

    ostream& print( ostream& out );
    bool bigger( Transaction& b ) const { return ID > b.ID; }

};

inline ostream& operator<<(ostream& out, Transaction& t){ return t.print(out); }
```

---

Figure 16.22. The data class: Transaction.

*Fourth box, Figure 16.22: Comparing two transactions.*

- The function `bigger()` is public and is called from `Charges::findMax()` to compare two transactions. The `Transaction` class is the expert on which transaction is bigger, so the `Charges` class delegates the comparison to it. This is a better design than having `Charges::findMax()` do the comparison itself, after using a getter function to get each of the money amounts.
- We made the function `bigger()` inline because it fits on one line. Being inline improves the time and space efficiency of short functions.

*Fifth box, Figure 16.22 and third box, Figure 16.23: Printing.*

- The `Transaction` class declaration is followed by an inline function definition for `operator<<`, the output operator. Operator extensions are beyond the scope of this book, and the student does not need to understand

---

These are the function definitions for the `Transactions` class in Figure 16.22

```
#include "trans.hpp"                                // File: trans.cpp

// -----
Transaction:: Transaction( int ID, float owes ) {
    this->ID = ID;
    this->owes = owes;
}

ostream& Transaction:: print( ostream& out ) {
    return out <<"[" <<setw(2) <<ID <<"]" " <<fixed <<setw(7) <<setprecision(2)
                <<owes <<endl;
}
```

---

Figure 16.23. Functions for the data class, `Transaction`.

this code at this time.

- Defining this method for `operator<<` enables us to use `<<` to print Transactions. The definition, itself, simply reaches inside the class to call the public print function in the Transaction class.
- `Transaction::print()` returns an `ostream&` result which is then returned again by `operator<<`. This allows us to use one line of code with a chain of calls on `<<` to print several things, including a Transaction and a newline.

## 16.7 What You Should Remember

### 16.7.1 Major Concepts

**Pointer operations.** To use pointers skillfully, several pointer operations must be understood:

- Direct assignment. To set a pointer, `p`, to point at a selected referent, `r`, write `p = r` for arrays or functions but write `p = &r` for structures or simple variables.
- Indirect assignment. To assign a new value, `v`, to the referent of pointer `p`, use `*p = v` (`v` cannot be an array or function).
- Direct reference. To copy pointer `p` into another pointer `q` write `q = p`.
- Indirect reference. To use the referent of pointer `p` in an expression, write `*p` for simple variables and array elements, `p->member_name` if `p` points at a structure, and simply use `p(...)` if it points at a function.
- Pointer increment. To make pointer `p` point to the next (or previous) slot of an array, write `++p` (or `--p`).
- Pointer arithmetic. To calculate the number of array slots between two pointers, `p` and `q` (where `q` points to a later slot), write `q - p`. Accessing an array element has two equivalent forms; `*(p+5)` and `p[5]` reference the same value. The latter is preferred.
- Pointer comparison. To test whether two pointers, `p` and `q`, refer to the same object, simply use `p == q`. To compare the values of the referents, compare `*p` to `*q` using an appropriate comparison operator or function (`==` or your own function for comparing two objects).

**Array and pointer semantics.** In a very strong sense, C doesn't really have array *types*. An array is simply a homogeneous, sequential collection of variables of the underlying type. We can do nothing with an array as a whole except initialize it and apply `sizeof` to it. When a pointer refers to an array, whether subscripted or not, it refers to only one slot at a time.

**Array allocation time.** If an array is declared with square brackets and a `#defined` length, its size is fixed at compile time and can be changed only by editing and recompiling the source code. At very little additional cost in terms of time and space, many programs can be made more flexible by using dynamic memory. The maximum expected amount of data is determined at run time, and storage is then allocated for an array of the appropriate size. Such an array is declared in the program as an uninitialized pointer variable, `p`. Then at run time, either `malloc()` or `calloc()` is called to allocate a block of memory, and the resulting starting address is stored in `p`. (Of course, this must be done before attempting to use `p`.)

**Resizable arrays.** It is possible to resize a dynamic array, making it either longer or shorter. If the array is lengthened, it may be reallocated starting at a new memory address. Therefore, when implementing your own growing data structures, you must copy all the data from the old block to the new block. Resizing an array is an appropriate technique for applications in which the amount of data to be processed cannot be predicted until after processing has begun.

**Recycling storage.** A program that uses dynamic memory is responsible for freeing that memory when no longer needed. Blocks used during only one phase of processing should be recycled by calling `delete` or `delete[]` as soon as that phase is complete. Some memory blocks remain in use until the end of the program. If all is working properly, such blocks will be freed by the system automatically when the program ends, and so the program should not need to free them explicitly. However, relying on some other program to clean up after yours is risky. It is better if every program frees the dynamic storage it allocates. Recycling memory is especially important if a program requests either several very large memory blocks or many smaller ones. Failure to free salvageable blocks can cause program performance to deteriorate. If the virtual address space becomes exhausted by many requests to allocate memory, and none to free it, there will be a fatal run-time error.

**Sorting.** Insertion sort is a simple sorting algorithm, implemented here using a double loop that moves data within the array. The sorting strategy is to pick up items from the unsorted part of the data set and insert them, in order, into the sorted portion. Insertion sort should not be used for large data sets, because it is very slow compared to other sorts such as quicksort, which is covered later in this book. However, it is considered the best sorting algorithm for small data sets (up to about 25 or 30 items on a modern computer).

### 16.7.2 Programming Style

**Data encapsulation.** Object-oriented languages such as C++ permit the programmer to define objects and data structures in a way that encapsulates everything about them. The `Flex array` implements this philosophy – it gathers together the information about an object into a structure and treats it as a single object.

**Pointers vs. subscripts.** A pointer can be used (instead of a subscript) to process an array. The scanning technique of using cursor and sentinel pointers is easy and very efficient when the array elements must be processed sequentially. When random access to elements is made, using a subscript is more practical with either an array name or a pointer to an array.

**Sorting preferences.** Always use the appropriate sorting algorithm for a particular situation. Speed usually is the deciding factor. For small data sets, the insertion sort performs well. For larger data sets, a fundamentally different sort is needed, like the quicksort algorithms discussed in Chapters 19 and 20.

**Coding idioms.** Errors with pointers are hard to track down because the symptoms may be so varied. Any computation or output that happens after the original error could be invalid because the program may be storing information in the wrong addresses and, thereby, destroying the data used by the rest of the program. This kind of error generally requires that the whole job be rerun and is doubly frustrating because the cause of the error can be hard to localize and, therefore, hard to fix. We address this problem by using coding idioms, presented throughout this chapter, that ensure all references to an array (either through subscripts or through pointers) refer to slots that are actually part of the array. A few of these are

- Initialize pointers to `nullptr`. Using a null pointer should cause the program to terminate quickly and make the problem easier to find.
- When referencing an element, `i`, of an array using a pointer, use the syntax `p[i]` rather than `*(p+i)`.
- Use an off-board sentinel pointer to mark the end of an array for a scanning loop.

### 16.7.3 Sticky Points and Common Errors

**Pointers out of bounds.** One danger of pointers in C++ is pointing at something unintended. Common errors are to use a subscript that is negative or too large or to increment a pointer beyond either end of an array and then attempt to use the pointer's referent. You cannot use a declaration to restrict a pointer to point at a legitimate array element. Also, C++ does not “enforce” the boundaries of an array at run time. In general, it does not trap pointer errors. An attempt to use a pointer that is out of bounds (or `NULL`) may cause the program to crash but, on some systems, may not be detected at all; the program will simply walk on adjacent memory values.

**Uninitialized sentinel.** Another common error is to use a pointer to process an array but forget to initialize the end pointer. The loop almost certainly will not terminate at the end of the array. After processing the actual array elements, the cursor will fall off the end of the array and keep going until, eventually, the program malfunctions or crashes. When you use a sentinel value, be careful to set your loop test correctly, depending on whether you are using an on-board or off-board pointer.

**Pointers and arrays.** Since an array name, without subscripts, is translated as a pointer to the first slot in the array, some books say that “an array is a pointer.” However, this clearly is not true. Since we can use an unsubscripted array name in almost any context where a pointer is expected,<sup>8</sup> we, accurately, could say that an unsubscripted array name becomes a pointer to the beginning of the array. But an array is not a pointer. A pointer requires only a small amount of storage, often 4 bytes.<sup>9</sup> In contrast, an array is a series of storage objects of some given type and can occupy hundreds of bytes. Conversely, a pointer certainly is not an array, it is not limited to use with arrays and cannot be used where an array is needed unless it refers to an array.

A common error of this sort is to attempt to use a string variable for input, but forget that an array must be declared to hold the characters that are read in. When the pointer is dereferenced, the program will usually crash.

**Wrong reference level.** The most common reason for pointer programs to fail is that the wrong number of ampersands or asterisks is used in an expression. Although most compilers give warning comments about mismatched types, they still finish the translation and generate executable code. Do not ignore warnings about pointer type errors.

**Incorrect referencing.** It also is common to write syntactically correct code with pointers that do not do what you intend. For example, you may wish to change the value of a pointer’s referent and, instead, change the value of the pointer itself. Also, you may forget to insert parentheses in the proper places, such as using `*p+5` rather than `*(p+5)` to access an array element. Finally, you may attempt to use a pointer of one data type to access the memory area in which data of another type are stored. Dereferencing such a pointer will result in a garbage value.

**Precedence.** When dereferencing a pointer using `*`, don’t be afraid to use parentheses around the dereferenced value when other operators are involved, thereby making sure that the proper precedence is both understood by you and used by the computer. Watch out for the precedence of `*` in an expression involving pointers, such as `*p++`. Depending on the situation, this expression might have been intended to increment the contents of the address in `p`, but at other times it might have been necessary to increment the contents of `p` and then use the new address. Use parentheses where needed for clarity.

**Pointer diagrams.** Errors sometimes stem from having an unclear idea about what the ampersand and asterisk operators really mean. The best way to avoid such trouble is to learn to draw diagrams of your pointers, objects, and intended operations, following the models at the beginning of this chapter. Having a clear set of diagrams can help reduce confusion when you begin to write code.

**Pointer arithmetic and logic.** It is not meaningful to use address arithmetic on a pointer unless the pointer refers to an array. Similarly, it is not meaningful to compare or subtract two pointers unless they point at parts of the same array.

#### 16.7.4 Vocabulary

These are the most important terms and concepts discussed in this chapter.

---

<sup>8</sup>For instance, we cannot do an assignment such as `arrayname = pointer`.

<sup>9</sup>This is true of many modern computers. However, some computers may have pointers of different lengths. Microprocessors in the Intel 80x86 family have two kinds of pointers, local (near pointers) and general (far pointers). The near pointer occupies only 2 bytes of storage. In the near future, computers may have such large memories that they will need more than 4 bytes for a pointer.

base type	pointer with subscript	scanning loop
referent	pointer assignment	dynamic allocation
<code>&amp;</code> (address of)	indirect assignment	constructor
<code>*</code> (indirect reference)	pointer arithmetic	destructor
<code>-&gt;</code> (dereference or select)	pointer comparisons	Flex array
<code>new</code>	off-board sentinel	insertion sort
<code>delete</code>	tail pointer (sentinel)	selection sort
<code>delete[]</code>	head pointer	reference level error
	scanning pointer (cursor)	

### 16.7.5 Self-Test Exercises

1. Given the array of values that follows, show the positions of the values after each pass of a selection sort that arranges the data in descending order:

21	4	13	17	24	8	15
----	---	----	----	----	---	----

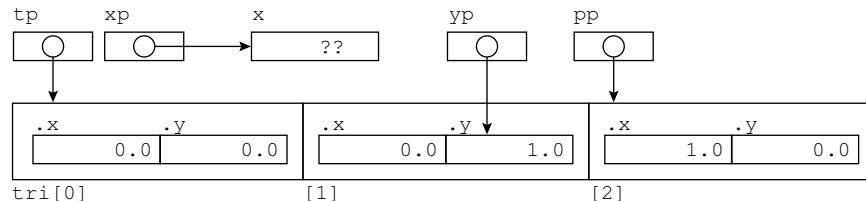
2. Repeat the previous problem on the data above using the insertion sort algorithm for ascending order.  
 3. Add the correct number of asterisks to the following declarations. Also add the correct number of ampersands in the initializers so that the declaration actually creates the object described by the phrase on the right. In the last item, replace the `???` by the correct argument.

- (a) `char a[12] = "Ohio";` An array of chars
- (b) `char b ;` An array of char pointers
- (c) `char p = b[0];` A pointer to the first slot in an array of char pointers
- (d) `char q = new char[12];` A pointer to a dynamically allocated array of 12 chars
- (e) `char s = new ??? ;` A pointer to a dynamically allocated array of 4 char pointers

4. You are given the declarations and diagram that follow:

```
typedef struct { double x, y; } PointT;
double x;
PointT tri[3];

double* xp;
double* yp;
PointT* pp;
```



- (a) Change the declaration of `tri` to include an initializer for the values shown in the diagram.
  - (b) Declare another pointer named `tp` and initialize it to point at the beginning of the array that represents the triangle, as shown.
  - (c) Declare and initialize `triEnd`, an off-board sentinel for the triangle.
5. Use the diagram in the preceding exercise to write the following statements:
- (a) Write three assignment statements that make the pointers `xp`, `yp`, and `pp` refer to the objects indicated.

- (b) Using only these four pointers (and not the variable names `tri` and `x`), write an assignment statement that copies the `x` coordinate of the last point of the triangle `tri` into the variable `x`.

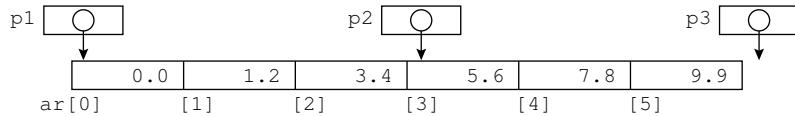
(c) Using the array name `tri` and a subscript, store the value 3.3 in the referent of `yp`.

(d) Using the pointer `tp` and no subscripts, print the `x` coordinate of the last point in the array.

6. Using the representation for triangles diagrammed and described in problem 5, write a function, `triIn()`, with no parameters that will read the coordinates of a triangle from the `cin` stream. Dynamically allocate storage for the triangle within this function and return the completed triangle by returning a pointer to the first point in the array. Use pointers, not subscripts, throughout this function.

### 16.7.6 Using Pencil and Paper

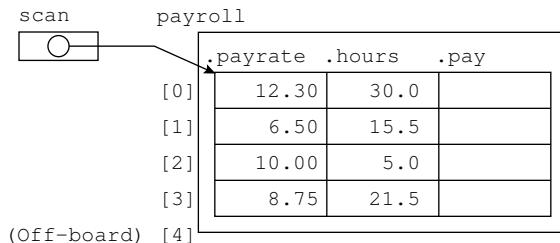
1. Show the changes that must be made to adjust the insertion sort program to sort numbers in descending order. Then use the data array first presented in Figure 16.5 to trace the steps of execution of the new insertion sort algorithm, as was done in Figure ??.
  2. Given the data structure in the following diagram, look at each of the statements. If the operation is valid, say what value is stored in `x` or in `p2` by each of these operations. If the operation is not valid, explain why. Assume that all numbers are of type `double` and all pointers are of type `double*`.



- (a) `x = *p1;`
  - (b) `x = p1 + 1;`
  - (c) `x = *(p2 + 1);`
  - (d) `x = *p2 + 1;`
  - (e) `p2 = p1[4];`
  - (f) `p2++;`
  - (g) `p2 = p1 + 1;`
  - (h) `p2 = p3 - p1;`

3. Given the data structure diagrammed in the next problem, write a type declaration and the set of variable declarations to construct (but not initialize) the payroll structure, the pointer `scan`, and pointers (not illustrated) `end` and `p10`, which will be used to process the structure.

4. Given the data structure in the following diagram, write a statement for each item according to the instructions given. Assume that all numbers are type `float`. Declare the pointers appropriately.



- (a) Set `scan` to point to the first structure in the array, as illustrated.
  - (b) Set `end` to be an off-board pointer for the array.
  - (c) Move `scan` to the next array slot.
  - (d) Set pointer `p10` to point at the slot with pay rate \$10.00.
  - (e) Give the last person a \$.50 pay raise.
  - (f) Calculate and store the pay for the person who is the referent of `p10`.

### 16.7.7 Using the Computer

1. Pointer selection.

Rewrite the selection sort program presented in Chapter 10 to use pointers for array searching and data accessing as in the insertion sort program in this chapter.

2. Sorting a poker hand.

This program asks you to begin implementing a program that runs a poker game. To start, you will need to define two enumerated types:

- (a) Represent the suits of a deck of cards as an enumeration (clubs, diamonds, hearts, spades). Define two arrays parallel to this enumeration, one for input and one for output. The input array should contain one-letter codes: {‘c’, ‘d’, ‘h’, ‘s’}. The output array should contain the suit names as strings.

- (b) Represent the card values as integers. The numbers on the cards, called *spot values*, are entered and printed using the following one-letter codes: {‘A’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘T’, ‘J’, ‘Q’, ‘K’}. These should be translated to integers in the range 1 … 13. Any other card value is an error.

- (c) Represent a card as class with two data members: a suit and a spot value. In the Card class, implement these functions:

- i. `Card::Card()`. Read and validate five cards from the keyboard, one card per line. Each card should consist of a two-letter code such as `3H` or `TS`. Permit the user to enter either lower-case or upper-case letters.

- ii. `void print()`. Display the cards in its full English form, that is, `9 of Hearts` or `King of Spades`, not `9h` or `KS`.

- (d) Represent a poker Hand as array of five cards. In the public part of the class, implement the following functions:

- i. `Hand::Hand()`. Read and validate five cards from the keyboard, one card per line. Each card should consist of a two-letter code such as `3H` or `TS`. Permit the user to enter either lower-case or upper-case letters.

- ii. `void sort()`. Sort the five cards in increasing order by spot value (ignore the suits when sorting). For example, if the hand was originally `TH, 3S, 4D, 3C, KS`, then the sorted hand would be `3S, 3C, 4D, TH, KS`. Use insertion sort and pointers.

- iii. `void print()`. Display the five cards in a hand, one card per line.

- (e) Write a main program to instantiate a Hand and test these functions.

3. Beginner’s poker.

The object of poker is to get a hand with a better (less likely) combination of cards than your opponents. The scoring combinations are listed in Figure 16.24, from the best possible hand to the worst. Define an enumerated type to represent these values and a parallel array for output. Start with a program that can read, sort, and print a poker hand, as described in the previous problem. Add these functions to begin implementing the game itself:

- (a) Write nine private functions, one for each scoring combination. Each function will analyze the five cards in this hand and return `true` if the hand has the particular scoring combination that the function is looking for. (Return `false` otherwise.) Some of these functions might call others.

- (b) `int handValue()`. Given a sorted hand, evaluate it using the scoring functions in Figure 16.24. Return the highest value that applies. For example, the hand `TS, JS, QS, KS, AS` is a royal flush, a straight flush, a flush, and a straight. Of these possibilities, royal flush has the highest value and should be returned.

- (c) Write a main program that reads two hands, calls the evaluation function twice, prints each hand and its value, then says which hand wins, that is, has a higher value. If two hands have the same value, then no one wins. (This is a slight simplification of the rules.)

4. Average students.

At Unknown University, the Admissions department has discovered that the best applicants usually

---

Value	Description
Royal flush	The suits all match and the spot values are T, J, Q, K, A
Straight flush	The suits all match and the spot values are consecutive
Four of a kind	The hand has four cards with the same spot value
Full house	The hand has one pair and three of a kind
Flush	All five cards have the same suit
Straight	The spot values of all five cards are consecutive
Three of a kind	The hand has three cards with the same spot value
Two pairs	The hand has two pairs of cards
One pair	The hand has two cards with the same spot value
Bust	Five cards that include none of the above combinations

---

Figure 16.24. Poker hands, high to low.

end up going to the big-name schools and the worst often do poorly in courses, so the school wants to concentrate recruitment efforts and financial aid resources on the middle half of the applicants. You must write a program that takes a file containing a list of applicants and prints an alphabetical list of the middle half of these applicants. (Eliminate the best quarter and the worst quarter, then alphabetize the rest.)

Define a class `Applicant` with data members of a name and a total SAT score. This can be done using a simple iterative technique. Following the example in Figures 16.22 and 16.23, define a constructor, and the functions `print` and `bigger` and `smaller`.

Read the data from a file named `apply.dat` into a Flex array of `Applicants`. Let  $N$  be the number of objects in the Flex array.

Then begin a loop that eliminates applicants two at a time, until only half are left. Within this loop, let `first` be a pointer to the first array element that is still included, `last` be a pointer to the last one, and `r` (the number remaining) be initialized to  $N/2$ . Each time around the loop, `first` is incremented, `last` is decremented, and `r` decreases by 2. At each step,

- (a) Using `smaller()` find the remaining applicant with the lowest SAT score and swap it with the applicant at position `first`. Then increment `first`. Then find the remaining applicant with the highest SAT score and swap it with the applicant at `last`. Then decrement `last`, subtract 2 from `r`, and repeat.
- (b) Quit when  $r \leq N/2$ . Return the values of `first` and `r`.

When only half the applicants are left, use any sort technique to sort them alphabetically using an insertion sort. Write the sorted list to a user-specified output file.

# Chapter 17

## Vectors and Iterators

This chapter gives an overview of the C++ standard template library, introduces the standard `vector` template class and explains how it works by analogy to the Flex array. Two applications are presented that rely on the dynamic nature of vectors.

### 17.1 The Standard Template Library—STL

A student of computer science must learn about data structures—these are classes that organize a collection of data for efficient storage and retrieval. With any particular type of data structure, the operations performed on it depend wholly on the structure and not on the data stored in it. STL is a library of pre-programmed data structures written by the best C++ developers, in the form of templates.

A template is an abstract class definition. When a real type is supplied as a parameter, the template code is *instantiated* with that type to create a real class definition that can then be compiled. The result is code that is customized for the given type parameter. For example, STL provides a template for a stack class. Suppose your program defines an Item class. Then you would create a stack of Items, named `s`, like this: `stack<Item> s;`

In the implementation of the Flex class, a `typedef` for an abstract type `BT` was used to instantiate the Flex for the desired base class. This use of `typedef` is an old C trick that is still useful, but it is not as powerful as a template implementation.

**The STL library.** STL was designed with extreme care so that it is complete and portable and as safe as possible within the context of standard C++. Among the design goals were:

- To provide standardized and efficient template implementations of common data structures, and of algorithms that operate on these structures.
- To produce correct and efficient code. The level of efficiency is greater than Java and Python are able to produce.
- To unify array and linked list concepts, terminology, and interface syntax. This supports plug-and-play programming and permits a programmer to design and build much of an application before committing to a particular data structure.

There are three major kinds of components in the STL:

- Containers manage a set of storage objects (vector, list, stack, map, etc.). Twelve basic kinds are defined, and each kind has a corresponding allocator that manages storage for it.
- Iterators are pointer-like objects that provide a way to traverse a container.
- Algorithms (sort, swap, make\_heap, etc.) use iterators to act on the data in the containers. They are useful in a broad range of applications.

In addition to these components, STL has several kinds of objects that support containers and algorithms including key-value pairs, allocators (to support dynamic allocation and deallocation) and function-objects.

Code	Meaning
<code>vector&lt;BT&gt; vc;</code>	Construct an empty vector of default size to hold <i>BT</i> objects.
<code>vc.push_back( bto )</code> <code>vc.pop_back()</code> <code>vc.clear()</code>	Insert an object at the end of the vector. Grow if necessary. Remove and discard the most recently inserted object. Remove all of the elements from the vector.
<code>k = (int) vc.size()</code> <code>c = (int) vc.capacity()</code> <code>vc[k] = bto</code> <code>bto = vc[k]</code> <code>bto = vc.front()</code> <code>bto = vc.back()</code>	Return the number of objects stored in <i>vc</i> (type <i>size_type</i> ) Return the number of slots currently allocated (type <i>size_type</i> ) Store a value in the contents of the <i>k</i> th slot of the vector. Return the contents of the <i>k</i> th slot of the vector. Return the first element in the vector. Return the last element in the vector.

Figure 17.1. Vector operations.

### 17.1.1 Containers

The C++ standard gives a complete definition of the functional properties and time/space requirements that characterize each container class. The intention is that a programmer will select a class based on the functions it supports and its performance characteristics. Although natural implementations of each container are suggested, the actual implementations are not standardized: any semantics that is operationally equivalent to the model code is permitted<sup>1</sup>.

**Member operations.** Some member functions are defined for all containers. These include: Constructors a destructor, traversal initialization: (`begin()`, `end()`), `size()` (current fill level), `capacity()` (current allocation size), and `empty()` (true or false).

Other functions are defined only for a subset of the containers, or for a particular container. For more information, go to [cplusplus.com](http://cplusplus.com) and click on Reference, then Containers, then `<vector>` or the name of another container.

## 17.2 The vector Class

The growing array is probably the most important data structure in use today. There are simple implementations, such as Flex array, and more powerful, general solutions such as the `ArrayList` class in Java and the C++ template class `vector`.

Both `FlexArray` and `vector` copy the data values into the data structure. This means that the base type must be copyable. Once copied, the `vector` “owns” the data object and will properly delete it at the end of the program. If the base type of the `vector` is a pointer type, pointing to a dynamic object, you must write a loop to delete those objects yourself.

It is essential to remember that any iterators in use are invalidated if the data structure grows or if elements are removed from the data structure.

A `vector` works the same way as a Flex array: it tracks its current capacity and the number of items stored in it. When those numbers are equal and more data is inserted, the `vector` “grows” in the same way that a Flex array grows: by doubling its capacity. The `FlexArray` is simpler and easier to use for those things it does implement. However, `vector` is a standard type that presents the same interface as the other STL container classes. Figure 17.1 lists the most important `vector` functions. Many functions are omitted here; for more information, consult a standard reference. In this chart, *BT* stands for the base type of the array, *bto* stands for an object of that type, and *vc* stands for the `vector` object.

The `vector` class is not as restricted as a Flex array. It implements more functions, including `swap()` and `sort()`. It can be used with any base type, whereas the implementation shown for Flex arrays cannot be used to store objects that contain dynamic parts and have a destructor that manages them.

<sup>1</sup>Big-Oh notation is used to describe performance characteristics. In the following descriptions, an algorithm that is defined as time  $O(n)$ , is no worse than  $O(n)$  but may be better.

Code	Meaning
<code>vector&lt; BT &gt;:: iterator p1, p2, pos;</code>	Create three iterators to point at vector elements.
<code>p1 = vc.begin();</code> <code>p2 = vc.end();</code> <code>p1++ , ++p2</code> <code>--p2 , p2--</code> <code>if (pos == vc.end())</code>	A iterator that points to the first element in the vector. An offboard iterator pointing to the first unused slot in vc. Move the iterator forward to the next array slot. Move the iterator backwards to the previous array slot. Test whether an iterator has reached the end of the vector.
<code>pos = find( p1, p2, bto );</code> <code>vc.sort (p1, p2)</code>	Return pointer to first copy of bto in the vector; p2 for failure. Sort the elements of the vector from p1 to but not including p2

Figure 17.2. Vector iterators.

**Iterators.** An iterator is a generalization of a pointer, and is used the way a pointer would be used. You can think of an iterator as a “smart pointer”. All STL containers have associated iterator types that “know about” the internal structure of the container and are able to move from the first element to the last element, hitting each element on the way. It is important to remember that any iterators in use are invalidated if the data structure grows or if elements are removed from the data structure. Figure 17.2 lists the essential `vector::iterator` functions.

### 17.2.1 Using the STL vector Class

In this example program (Figure 17.3) we create a vector of ints and an iterator for it, populate the vector, sort it, search it, remove some items and print it. The STL vector functions used are the default constructor, `push_back()`, `pop_back()`, `[ ]`, `size()`, `sort()`, `find()`, `erase()`, `front()`, `begin()`, and `end()`.

#### Notes on Figure 17.3: Vector demo program.

##### *First box: The environment.*

- The vector header file is needed to use the vector class.
- The algorithm header file is needed to use the sort and find functions.

##### *Second box: A global print function.*

- This is just a demo program, not a proper OO design. There is no class declaration. `main()` instantiates a vector and uses it. So there is nowhere to put a print function except in the global namespace. Because it is not in a class, this function must take the vector as a parameter. It is passed by reference (`&`) to avoid copying the data structure.
- The type qualifier `const` means that the print function will not modify the vector.
- Subscripts are used in this function to access the elements of the vector. Iterators are used for the same purpose in box 4.
- We use `vec.size()` to find our how many ints are stored in the vector, then we use the information to control the printing loop.
- An ordinary `for` loop was used in this example to illustrate calls on `size()` and to give better quality output. However, it is not the easiest or most modern way to write the print loop. Here is the modern alternative, using a for-each loop:

```
void print(const vector<int>& vec) // print the elements of the vector
{
    for (int k : vec) cout << k << endl;
    cout << "--- done ---\n";
}
```

Read this code as “for each int (call it k) in vec (the vector of ints), print k”.

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void print(const vector<int>& vec) { // print the elements of the vector
    int count = vec.size();
    for (int idx = 0; idx < count; idx++)
        cout << "Element " << idx << " = " << v[idx] << endl;
    cout << "--- done ---\n";
}

int main( void ) {
    vector<int> ages;           // create a vector of int's
                                // insert some numbers in random order
    ages.push_back(11);        ages.push_back(82);      ages.push_back(24);
    ages.push_back(56);        ages.push_back(6);

    cout << "Before sorting: " << endl;
    print(ages);               // print vector elements
    sort(ages.begin(), ages.end());
    cout << "\nAfter sorting: " << endl;
    print(ages);               // print elements again

    cout << "First element in vector is " << ages.front() << endl;
    int val = 3;                // search the vector for the number 3
    vector<int>::iterator pos;
    pos = find(ages.begin(), ages.end(), val);
    if (pos == ages.end())
        cout << "\nThe value " << val << " was not found" << endl;

    cout << "\nNow remove last element and element=24 " << endl;
    ages.pop_back();

    // remove an element from the middle; the hole will be closed up.
    pos = find(ages.begin(), ages.end(), 24);
    if (pos != ages.end()) ages.erase(pos);
    print(ages);               // print 3 remaining vector elements

    return 0;
}

```

Figure 17.3. Vector demo program.

***Third box: Creating and filling the vector.***

- To instantiate a vector, we must supply a base type enclosed in angle brackets. This type can be any defined primitive type or class. Initially, this vector is empty.
- We put five elements into the vector (each goes at the end).
- Using `push_back()` is the only proper way to insert more data into a vector. Only `push_back()` will cause the structure to grow.

***Fourth box: Before and after sorting.***

- `ages.begin()` is an iterator pointing at the first item in the vector. `ages.end()` is an offboard iterator pointing at the vector slot not occupied by data. The functions `begin()` and `end()` are defined on all containers.
- Calling `sort(ages.begin(), ages.end())` sorts the data from the beginning to the end. A smaller portion of the data can be sorted by supplying iterators to slots in the middle.
- The contents of the vector are printed before and after sorting. Here is the output, condensed into two columns. Read the left column first.
- The contents of the vector are printed before and after sorting. Here is the output, condensed into two columns. Read the left column first.

Before sorting:	After sorting:
Element 0 = 11	Element 0 = 6
Element 1 = 82	Element 1 = 11
Element 2 = 24	Element 2 = 24
Element 3 = 56	Element 3 = 56
Element 4 = 6	Element 4 = 82
--- done ---	--- done ---

***Fifth box: Accessing and searching a vector.***

- The call on `ages.front()` gets the first element in the vector but does not erase it from the vector.
- We prepare for searching by declaring an iterator named `pos` of the right kind for `vector<int>`.
- Like `sort()`, a call on `find()` takes two iterator parameters to mark the beginning and the end of the part of the vector to be searched. The third parameter is the value to search for.
- The result of `find()` is an iterator pointing at the desired value, if it exists, or a copy of the second parameter if the value is not in the array.
- In this box, the sought-for value is not on the vector, so the call `pos = find(ages.begin(), ages.end(), val)`, sets the iterator `pos` to point at `ages.end()`, the first vector slot not occupied by data. We test for this condition and the error comment is printed.
- The output:

```
First element in vector is 6
The value 3 was not found
```

***Sixth box: Removing data from a vector***

- We can remove the last vector element by calling `pop_back()`. This function is very efficient: it simply decrements the vector's counter for data items.
- We can also remove any element that an iterator is pointing at. Here, we use `find()` to set an iterator to the slot containing 24, then use `erase()` to remove the 24. This operation is slow because the `remove()` function must shift each element that follows the 24 one slot to the left to fill up the hole.
- A final call on `print()` verifies the actions described here. The output is:

```
Now remove last element and element=24
Element 0 = 6
Element 1 = 11
Element 2 = 56
--- done ---
```

---

### The Pleasant Lakes Club

A group of neighboring families got together, bought two cottages on a lake, and incorporated as the Pleasant Lakes Club. Their bylaws say that there must be no more than 20 families in the club, so that each family can use a cottage one week per summer. Although the club currently has fewer than 20 families, they are willing to buy another cottage if the membership grows.

#### **The software.**

The club secretary maintains a file of member information and decided to write a program to help with that task. He has found that the sizes of families and the length of their names vary dramatically, so he decided that his program will use dynamic allocation (vectors and strings) to organize this information. When a new family joins, a Family object is created and pushed onto the membership vector. The family object contains some family data and a vector that stores the name of each person in the family.

---

**Figure 17.4. Problem Specification.**

### 17.3 A 3-D dynamic object.

As a first example of the application of vectors, we implement the software described in Figure 17.4. Our solution creates a vector of objects, where each object contains a vector of strings. We use this data structure to model the membership of a small club. This is a 3-dimensional dynamic data structure: (1) the vector of families can grow as large as needed, (2) the vector of names in each family can grow during input, and (3) each name (a string) can be as long as needed.

This program is a preliminary version of a membership-management tool that intended to maintain a club's membership database file. The use of vectors and strings makes it easy to handle all the variability in the data.

The overall strategy is to read in the current file of members and display them so that the secretary can see who is there and who is not. Then, if he has a new member family, the family information can be typed in, stored in the vector, and ultimately written back out to the file.

Input and output for variable-length objects is significantly trickier than input for a flat database. The main program and the functions in the FamilyT class must work closely together to handle the nature of the data file and the possible kinds of input errors. They share responsibility for detecting end-of-file and for interacting with the user.

The input file and the output file have the same name to make it easy to use the system over and over. After reading the input, the file is closed and renamed as a backup file. Then a new file of the same name is opened for output. By doing this, we avoid the danger of losing the data entirely if the program crashes while the file is open for output.

#### Notes on Figure 17.5: The Pleasant Lakes Club.

##### *First box: The environment.*

- By including `family.hpp` we are including `tools.hpp` and, indirectly, including `<vector>`, `<string>`, `<iostream>`, `<fstream>`, `<iomanip>`, several C header files, and the `namespace` command. Therefore we do not need to list all of these things at the top of this program.
- File names are defined at the top of the program and used throughout. This is good practice. Names for both the backup file and the output file are provided.

##### *Second and fourth boxes of main program: The input and output files.*

- The second box opens an input file in the ordinary way. The third box uses the file, and the fourth box closes it.
- It is always good to close a file as soon as the program is done with it. In this case, however, it is not just good, it is necessary. We want to rename the input file as a backup file, and we want to reuse the name for a different file. It needs to be closed first.
- The `rename()` function is from the C standard I/O library (`<cstdio>`). If a backup file already exists, it will be overwritten.
- After renaming the input file, we open a new output file with the original file name. If a file already exists by this name in the directory, it will be overwritten.

```

#include "family.hpp"
#define FILE "famFile.txt"
#define BAKFILE "famFile.bak"

int main( void )
{
    char reply;                                // For a work loop.

    ifstream iFams( FILE );
    if( !iFams.is_open() ) fatal( "Cannot open " FILE " for input." );

    cout <<"\n Pleasant Lakes Club Membership List \n";
    vector<FamilyT> club;
    for(;;) {
        FamilyT f;                            // Create an empty family.
        f.realize( iFams );                  // Read 1 family's data from the file.
        if (iFams.eof()) break;
        f.print( cout );                    // If data, display and add to club vector.
        club.push_back( f );
    }

    cout <<"\n----- Done reading club members -----\";

    iFams.close();
    rename( FILE, BAKFILE );
    ofstream oFams( FILE );
    if( !oFams.is_open() ) fatal( "Cannot open " FILE " for output." );

    for(;;) {
        cout <<" Do you want to enter a new family? (y/n): ";
        cin >>reply;                         // Read.
        cin.ignore(1);                      // Remove newline from stream.
        if (tolower( reply )!= 'y') break;
        FamilyT f;
        f.input();                           // Interactive input.
        club.push_back( f );
    }

    // Finally, write it all out again. -----
    for (FamilyT f : club) f.serialize( oFams );
    oFams.close();
    cout <<" Data is in file " <<FILE <<'\\n' <<" Normal termination." <<endl;
}

```

Figure 17.5. The Pleasant Lakes Club.

---

```

#include "tools.hpp"

class FamilyT {
private:
    int n = 0;                      // Number of people in this family.
    string fName;                   // The family's last name (father or mother).
    vector<string> fam;           // Dynamic array of strings.

public:
    FamilyT() = default;

    void input();                  // Input a family interactively.
    void print( ostream& out );   // Display a family, formatted nicely.

    void realize( istream& in );   // Read a family from a file.
    void serialize( ostream& out ); // Write a family to a file.

}

```

**Figure 17.6.** The FamilyT class.

**Third box of main program: File input.**

- To begin, we create an empty vector of families and, inside the loop a single empty family, `f`, with a default initialization.
- Having `f` allows us to call functions from the FamilyT class. First, we call `realize()`. We use this name for a function when it initializes a data structure by bringing in data from backup storage (in this case a disk file).
- All the details of reading the data are inside the FamilyT class, which is the expert on how to read the data for a family. Each time it is called, it reads one family from the stream.
- The end of file flag will be set during `serialize()` when we try to read another family that is not there. `Serialize()` does test for eof. However, we also need to know about eof in this function that we can end the loop.
- When eof happens in a stream, the stream eof flag is set and stays set until the stream is closed. We can test for eof anywhere in the program., even in a different function in a different module.
- If there was no `eof()`, then `f` has real data in it, so we call `FamilyT::print()` to display the data on the screen, then push the new family object into the vector.
- During the `push_back()`, the object is copied from the local variable `f` into the vector. The next time around the loop, `f` is recreated and reinitialized. We are never writing new family data on top of a former family.

**Fifth box: Interactive input.**

- This is written as a query loop. The user is asked each time around the loop whether he wishes to enter another family.
- After reading the single-character answer to the query, a newline character is left unread in the stream buffer. It must be removed; `ignore(1)` does the job.
- Inner box: Interactive input. Again, the FamilyT class hides all the details of reading the input for a new family. When the function returns, `f` has been initialized and can be pushed into the vector.

```

#include "family.hpp"

void FamilyT:: input() {
    string name;                                // For input of family's name.
    cout <<"\n Please enter the surname of the head of the family: ";
    getline( cin, fName );

    cout <<"\n Now enter the names of family members.\n The surname"
        <<" may be omitted if same as family's name.\n"
        <<" Enter RETURN to quit.\n";

    for (;;) {                                    // Read & install new names.
        cout <<" > ";
        getline( cin, name );
        if (name.length() == 0) break;
        fam.push_back(name);
    }

    n = fam.size();
}

// -----
void FamilyT:: print( ostream& out ) {
    int n = fam.size();
    cout <<"\n The " <<fName <<" family has " <<n <<" members.\n";
    for (string s : fam) cout <<'t' <<s <<'\n';
}

// -----
void FamilyT:: realize( istream& in ) { | string name;
    in >> n >> ws;
    if (!in.good()) {
        if (in.eof()) return;
        else fatal( "Family size is corrupted." );
    }

    getline( in, fName );
    for (int k=0; k<n; ++k) {
        getline( in, name );
        if (!in.good()) fatal( "Family name is missing." );
        fam.push_back( name );
    }
}

// -----
void FamilyT:: serialize( ostream& out ) {
    out << fam.size() <<" " <<fName <<endl;
    for (string s : fam) out <<s <<'\n';
}

```

Figure 17.7. The FamilyT functions.

**Notes on Figures 17.6 and 17.7: The FamilyT class.**

**First box: Data members.**

- The number of people in the family is needed to be able to read the database back in after writing it to a file.
- This is a prototype implementation – in a more developed implementation there would be data members to store a family's contact information.
- The vector of strings in this prototype would become a vector of PeopleT in a more developed version, so that a variety of personal information could be stored.

**Second box: the constructor.**

- A constructor without parameters is needed because `main()` FamilyT variables. This constructor performs a default initialization, setting the name to an empty string and the vector to an empty vector.

**Third box of Figure 17.6 and first box of Figure 17.7: Interactive input.**

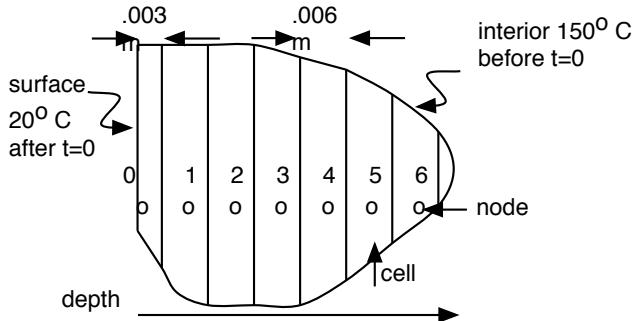
- This function reads the data for one family from the keyboard and stores it in the implied parameter (`this`).
- The calls on `cout <<` show that effort and thought have gone into communicating with the user.
- An input prompt, `>`, tells the user that input is expected. A short prompt like this is all you need. Long, wordy prompts repeated every time around a loop cause visual clutter and are not helpful.
- Inner boxes: We call `getline()` twice to read strings that might have embedded spaces. The output file was planned so that reading these strings would be easy: they are the last thing on each line.
- `getline()` reads and stores characters up to the end of the line. It removes the newline character from the stream and discards it. A newline is not stored in the string.
- If the user types RETURN instead of a name, the length of the string that is read will be 0. This is an easy way to end an input loop.

**Third box of Figure 17.6 and second box of Figure 17.7: Interactive output.**

- This function formats the data for humans to read. The `serialize()` function formats it for the computer to read on the next run. These formats are significantly different. The screen output for a family has a heading consisting of the family name and number of members.
- A for-each loop is used to print out the contents of the `fam` vector. Read this line as “for each family (call it `s`) in `fam`, display `s` on `cout`”.
- Use the for-each loop in preference to a for loop or a loop with iterators whenever you need to process all of the data stored in a vector.

**Fourth box of Figure 17.6 and third box of Figure 17.7: File input.**

- This function reads the data for one family from the input stream and stores it in the implied parameter (`this`).
- You simply cannot use a “while not end of file” loop to do this job because the data structure is variable length in three dimensions.
- The first item to be read for each family is the number of family members. The end-of-file flag will not go on in the stream until an attempt has been made to read this number for a family that does not exist. This code distinguishes between the normal end of file and an error caused by damage to the contents of the input file. Corrupted data will terminate the run. EOF will cause a return to the caller.



**Figure 17.8. Heat conduction in a semi-infinite slab.**

- An input prompt, `>`, tells the user that input is expected. A short prompt like this is all you need. Long, wordy prompts repeated every time around a loop cause visual clutter and are not helpful.
- Inner box : The first `getline()` reads the family name. The second one (in the loop) reads an individual name. The stream could be not good after either read, but it is adequate to test just once because the two calls are so close together and nothing will happen to the stream state after the first error.
- An effort was made to provide a meaningful error comment, but a variety of errors could cause a failure here and no error comment will be right for all of them.
- As always, if control gets past the input and error checks, we have good data and it is stored in the vector.

***Fourth box of Figure 17.6 and fourth box of Figure 17.7: File output.***

- To “serialize” a data structure means to write it to backup storage. This `serialize()` function is very much like the `print()` function, but simpler because the file output format is simpler than the interactive output format.

## 17.4 Using Vectors: A Simulation

The technique we use in the next example, a simulation, involves two pointers, `old` and `new`, that switch back and forth between pointing at two vectors, first referring to one, then to the other. This lets us represent an indefinite series of arrays, where the array values in `new` at each time step are derived from the ones in `old`. There is no need to allocate a long series of separate arrays or constantly move the data values from one array to another; we just **swap pointers** (or the addresses in them).

### 17.4.1 Transient Heat Conduction in a Semi-Infinite Slab

Problems that involve changes over time in a property of a solid or fluid often can be solved by analytical techniques when the geometry, boundary conditions, and material properties are simple. This is the preferred method, because a valid result can be determined at any continuous point inside the material at any time. However, when an analytical solution is not possible, numerical techniques can give an approximate solution at discrete points inside the material at specific times.

Transient heat conduction in a solid slab forms a class of problems suitable for **numerical approximation**. A slab, as shown in Figure 17.8, is divided by imaginary boundaries into equal-sized regions called *cells*. For each cell, the temperature is determined at a discrete point called its *node*. An energy equation is used to derive a formula for the temperature at the cell’s node, for each cell at each time step, in terms of the temperature at each of the surrounding nodes on the previous time step. This **finite-difference equation** (a form of the heat conduction equation) for each cell can be modeled by a computer program.

For the specific example in Figure 17.8, the slab’s initial uniform temperature is 150°C. It has a thermal diffusivity of  $6 \cdot 10^{-7} \text{ m}^2/\text{s}$ . Suddenly, at time  $t = 0$ , it is exposed to a cooling liquid so that the surface is instantly cooled to 20°C, where it remains throughout the process.

We want to determine the temperature at various depths below the surface of the slab as time passes. As mentioned, this process can be represented by a partial differential equation. The numerical approach to solving it assumes that the slab can be split into cells and that each cell has the same uniform temperature throughout as at its node, which is at the midpoint of the cell. The nodes are labeled 0, 1, 2, 3, ... beginning at the surface and are spaced 0.006 meter apart. The set of finite-difference equations used to compute the temperature  $T$  at nodes 1, 2, 3... is

$$T_m^{t+1} = \frac{1}{2} (T_{m-1}^t + T_{m+1}^t) \quad \text{for } m = 1, 2, 3, \dots \quad \text{and } t = 0, 1, 2, 3, \dots$$

where  $m$  denotes the node and  $t$  is the number of elapsed time steps, each corresponding to a 30-second interval. That is,  $t = 0$  at time 0,  $t = 1$  after 30 seconds,  $t = 2$  after 60 seconds, and so on.

At time 0 in the example, the cooling source at 20°C is applied at the edge of the slab, which initially is at a uniform temperature of 150°C. Therefore, the temperatures of the first four nodes at time 0 become  $T_0^0 = 20$ ,  $T_1^0 = 150$ ,  $T_2^0 = 150$ ,  $T_3^0 = 150$ . At the next time step (30 seconds later),  $t = 1$  and we can compute the nodal temperatures as

$$\begin{aligned} T_0^1 &= 20 \\ T_1^1 &= \frac{1}{2} (T_0^0 + T_2^0) = \frac{1}{2} (20 + 150) = 85 \quad \text{at a depth of 0.006 m} \\ T_2^1 &= \frac{1}{2} (T_1^0 + T_3^0) = \frac{1}{2} (150 + 150) = 150 \quad \text{at a depth of 0.012 m} \\ T_3^1 &= \frac{1}{2} (T_2^0 + T_4^0) = \frac{1}{2} (150 + 150) = 150 \quad \text{at a depth of 0.018 m} \end{aligned}$$

As time passes, the cooling effect penetrates deeper into the slab. The next time step corresponds to 60 seconds; at that time, the nodal temperatures are

$$\begin{aligned} T_0^2 &= 20 \\ T_1^2 &= \frac{1}{2} (T_0^1 + T_2^1) = \frac{1}{2} (20 + 85) = 85 \\ T_2^2 &= \frac{1}{2} (T_1^1 + T_3^1) = \frac{1}{2} (85 + 150) = 117.5 \\ T_3^2 &= \frac{1}{2} (T_2^1 + T_4^1) = \frac{1}{2} (150 + 150) = 150 \end{aligned}$$

After 90 seconds, node 3 will begin to cool. Eventually, if the cooling source remains constant, the cooling effect will reach the end of the slab, and the entire slab will approach a steady-state temperature of 20°C.

### 17.4.2 Simulating the Cooling Process

Vectors and iterators are used to implement this process. A call chart for this application is shown in Figure 17.9. The program is given in Figures ?? through 17.14.

#### Notes on Figures ??: A heat flow simulation.

**Main is simply the boss.** This is what a main program should look like if there are no files involved – all the details of the object model and the process are hidden in the class.

- Include the header file for the primary class.
- Instantiate that class.
- Call its functions to get the work done and the results printed.

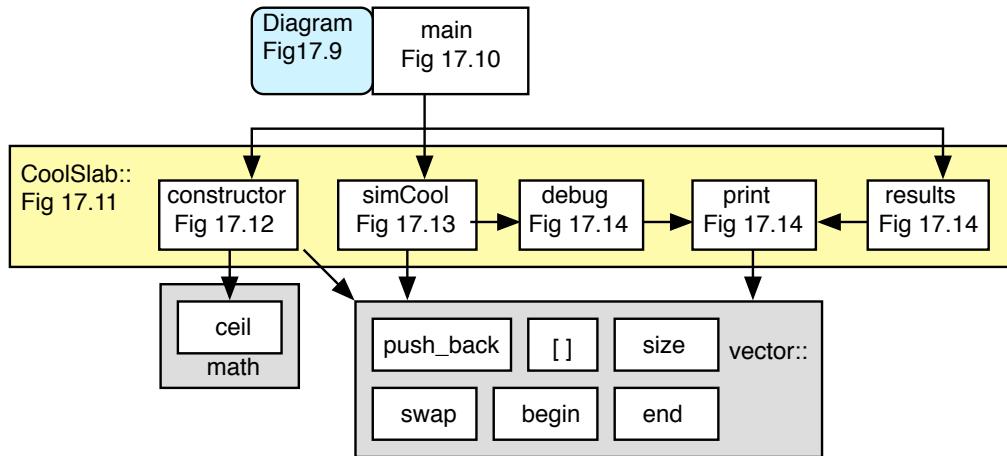


Figure 17.9. A call chart for the heat flow simulation.

Notes on Figures 17.11 and 17.14: The simulation class.

*First box of Figure 17.11: simulation parameters.*

- The first three variables are used to hold the simulation parameters entered by the user.
- All parts of the simulation depend on the depth at which the temperature is to be monitored. We use this depth to calculate `slot`, the number of the node at the desired depth, which then controls vector growth and looping.
- `p` counts the simulation steps. It is needed for output and to limit the length of the simulation in the event that the user enters unrealistic parameters.

*Second box of Figure 17.11: vectors and iterators.*

- We need two vectors to model heat in the cells of a slab. The vector named `old` contains the information for time step  $t$ , and `next` represents time step  $t + 1$ . Each value in `next` is calculated from two values in `old`. This process cannot be done in one array – it requires two.
- An iterator is defined for `next`. It will be used for printing.
- `p` counts the simulation steps. It is needed for output and to limit the length of the simulation in the event that the user enters unrealistic parameters.

*Third box of Figure 17.11 and Figure 17.12: Constructor and destructor.*

- Sometimes a main program handles all user interaction. In this program, however, the input will technical data about the simulation, so it belongs in the Slab constructor, which is the expert on simulations.

This code calls the functions from Figures 17.11, through 17.14.

```

#include <iostream>
#include "slab.hpp"

int main( void ) {
    cout <<"\nSimulation of Heat Conduction in a Slab\n";
    CoolSlab s;
    s.simCool();
    s.results( cout );
}
  
```

Figure 17.10. A heat flow simulation.

- The Slab constructor interacts with the user to input the parameters for the simulation. It uses those parameters to initialize the data members and the two vectors.
- Because all parts of the simulation depend on the depth at which the temperature is to be monitored, limits are defined for that depth and enforced by an input validation loop. If the depth value is large, the number of iterations needed to reach the goal temperature also will be very high and may exceed the predefined iteration limit, **MAX\_STEPS**. If this happens, the simulation will be halted prior to reaching the goal. Therefore, as a practical matter, we limit the observation depth to 0.25m and **#define** this constant (**DEPTH\_LIMIT**) at the top of the program. Obviously, negative values are rejected.
- Based on a valid depth, the slot number is calculated, according to the specification.
- The **ceil()** function from the C math library takes a double parameter and rounds it to the nearest greater integer.
- Second box: Reasonable bounds for the other inputs depend on the real process being simulated and cannot, in general, be established. So there are no validation loops for the three temperatures.
- Third box: After reading the temperatures, the two vectors can be initialized. To begin, we simply need the first two slots initialized. One new value will be added to the end of each vector on each simulation step. When these later elements are pushed into the vector, they will be initialized to the **initTemp**.
- Here is an example of the output produced by the constructor:

```
Simulation of Heat Conduction in a Slab
Enter depth to monitor (up to 0.25 meters): .055
```

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
using namespace std;

#define MAX_STEPS 1000
#define DEPTH_LIMIT .25

class CoolSlab {
private:
    double initTemp, goalTemp;           // Beginning and ending temperatures.
    double surf;                         // Temperature at surface of slab.
    double depth;                        // Depth at which to monitor temperature.
    int slot;                            // Array subscript at given probe depth.
    int p;                               // The actual number of simulation steps.

    vector<double> old;                // The initial conditions.
    vector<double> next;               // The next step of the simulation.
    vector<double>::iterator nIt;

public:
    CoolSlab();
    ~CoolSlab() = default;

    void simCool();

    void results( ostream& out );
    void debug( ostream& out );
    void print( ostream& out );

};
```

Figure 17.11. The Slab class.

```

CoolSlab:: CoolSlab() {
    do {
        cout <<"Enter depth to monitor (up to " <<DEPTH_LIMIT <<" meters):  ";
        cin >> depth;
    } while (depth<0 || depth > DEPTH_LIMIT);
    slot = ceil( (depth-.003) / .006 );

    cout <<"Enter initial temperature of the slab:  ";
    cin >>initTemp;

    cout <<"Enter target temperature for depth " <<depth <<":  ";
    cin >>goalTemp;

    cout <<"Enter temperature of cold outer surface:  ";
    cin >>surf;

    // Initialize first two slots of both vectors to the initial conditions.
    old.push_back( surf );
    next.push_back( surf );
    old.push_back( initTemp );
    next.push_back( initTemp );
}

```

**Figure 17.12.** The Slab constructor and destructor.

```

Enter initial temperature of the slab: 100
Enter target temperature for depth 0.055: 85
Enter temperature of cold outer surface: 0

```

- The destructor: Although we are using dynamic allocation, it is encapsulated within the `vector` class, which takes all responsibility for allocating and freeing the memory. The client class (Slab) does not need to free anything. We emphasize this by explicitly defining a default do-nothing destructor.

**Fourth box of Figure 17.11 and Figure 17.13: the simulation.** The `simCool()` function will simulate the heat flow process at successive time steps as the slab cools. It will stop when the selected node reaches a specified temperature or when the number of steps reaches the limit (`MAX_STEPS`). On each iteration, the new temperature at each node is computed, based on the temperatures of its neighboring nodes during the preceding time step.

- First box: At each time step during cooling, the leading edge of coolness progresses on slot to the right. Therefore, to begin each iteration, we extend the data in the vectors by one slot, initialized to the initial temperature of the slab. Even though we add an element to the vector every time around the loop, the vector only lengthens itself occasionally, when its available storage is full. This growth process is invisible to the client program.
- Second box: Walk down the vectors using `k` to subscript both of them. Calculate the next value by averaging the two old values to its left and right. This simple formula is a good model of what actually happens during cooling!
- During development, it was necessary to see what was happening in the arrays. So we defined a debug function to make that easy. Using a function removes the details of debugging from the flow of the main logic.
- Third box: The end of the simulation. When the temperature at subscript `slot` is  $\leq$  the goal temperature, the simulation is finished. The part of the test that follows the `&&` tests this and breaks out of the simulation loop when it happens.

The first part of the test in this `if` statement acts as a *guard* for the second part, which is the termination condition. However, in the early stages of the simulation, `next[slot]` may not exist yet, or it may exist but

---

This function is called by the main program in Figure ?? to perform the steps of the simulation.

```
// -----
void CoolSlab:: simCool() {
    for (p=1; p<=MAX_STEPS; ++p) {
        old.push_back( initTemp );      // Lengthen initialized parts by 1 slot.
        next.push_back( initTemp );     // to prepare for next simulation step.

        //Calculate the next set of conditions based on the old ones.
        for (int k = 1; k <= p; ++k) {
            next[k] = (old[k-1] + old[k+1]) / 2.0;
        }

        debug(cout);
        if (slot < next.size() && next[slot] <= goalTemp ) break;

        next.swap( old );
    }
}
```

---

**Figure 17.13.** `simCool()`: doing the simulation.

not have any valid data pushed into it. We must avoid testing vector slots that do not exist or do not hold valid data! Therefore, we compare `slot` to the `next.size()` before trying to access the data at `next[slot]`. If `slot > next.size()`, the if statement fails and we stay in the simulation loop<sup>2</sup>.

- Fourth box: The swap. At the end of each time step, we are done forever with the values stored in `old`, and `next` will become the basis for computing the next iteration. We also need a vector to hold the temperatures we calculate on the next iteration. The solution is called **swing buffering**: swap the old and the next so that the current next becomes the new old and the current old vector is reused for more calculations.

The vector class provides a function `swap()` that swaps the values of two vectors and does it efficiently, by swapping pointers, not copying all the data.

- The iterations continue until the goal temperature is reached or the maximum number of steps is exceeded. If the goal is attained, the program breaks out of the loop and returns to `main()`, which prints the final temperature values.

**Notes on Figures 17.14: Output functions for the simulation.** We want two kinds of output: (1) a brief format, appropriate for debugging, that will print the contents of a vector at one time step and (2) a full version giving the results of the simulation including the final temperatures reached. The common part of the two formats is printing the vector at the end of one time step. Thus, we define a `print()` function to do the common part, and two other output functions to give the two views we need.

- The `debug()` function. For debugging, we want to see a sequence of iterations, each numbered consecutively. So the `debug()` function prints the iteration number and a visual divider to make the output more readable. Here is a sample output; you can see the size of the vector increasing.

```
1. -----
[ 0]= 0.000 [ 1]= 50.000 [ 2]=100.000

2. -----
[ 0]= 0.000 [ 1]= 50.000 [ 2]= 75.000 [ 3]=100.000

3. -----
[ 0]= 0.000 [ 1]= 37.500 [ 2]= 75.000 [ 3]= 87.500 [ 4]=100.000
```

---

<sup>2</sup>Remember that logical operators are evaluated left to right using lazy evaluation.

The `results()` function is called from `main()` in Figure ??; `debug()` is called from `simCool()` in Figure 17.13. Both of these functions delegate the common parts of the job to `print()`.

```
// -----
void CoolSlab:: debug( ostream& out ){
    cout << '\n' << setw(3) << p << ". ----- \n";
    print( cout );
    cout << endl;
}

// -----
void CoolSlab:: results( ostream& out ){
    cout << "\nTemperature of " << next[slot] << " reached at node " << slot
        << "\n\tin " << p << " seconds (= " << fixed << setprecision(2)
        << p/60.0 << " minutes or " << p/3600.0 << " hours).\n"
        << "\nFinal nodal temperatures after " << p << " steps:\n";

    print( cout );
    cout << "\n\n";
}

// -----
void CoolSlab:: print( ostream& out ){
    int k = 0;

    out << fixed << setprecision(3);
    for (double d : next) {
        out << "[" << setw(2) << k << "]=" << setw(7) << d;
        if (++k % 5 == 0) out << "\n";           // Newline after every five items.
    }
}
```

Figure 17.14. `Print()`, `results()`, and `debug()`.

- The `results()` function. At the end of execution we want to see summary information and the final results of the simulation. We print the summary information and call `print()` to print the rest. This sample output corresponds to the parameters given earlier:

```
Temperature of 84.614 reached at node 9
in 39 seconds (= 0.65 minutes or 0.01 hours).
```

```
Final nodal temperatures after 39 steps:
[ 0]= 0.000 [ 1]= 12.537 [ 2]= 25.074 [ 3]= 36.417 [ 4]= 47.760
[ 5]= 57.041 [ 6]= 66.322 [ 7]= 73.181 [ 8]= 80.041 [ 9]= 84.614
[10]= 89.187 [11]= 91.931 [12]= 94.675 [13]= 96.152 [14]= 97.630
[15]= 98.341 [16]= 99.052 [17]= 99.357 [18]= 99.662 [19]= 99.778
[20]= 99.893 [21]= 99.932 [22]= 99.971 [23]= 99.982 [24]= 99.993
[25]= 99.996 [26]= 99.999 [27]= 99.999 [28]=100.000 [29]=100.000
[30]=100.000 [31]=100.000 [32]=100.000 [33]=100.000 [34]=100.000
[35]=100.000 [36]=100.000 [37]=100.000 [38]=100.000 [39]=100.000
[40]=100.000
```

Note that slots 28 – 40 all print as 100.000 degrees. This is an artifact of the print formatting (three places of precision). The number in slot 40 is actually 100.000. The numbers in slots 28 – 39 are all between 99.999 and 100, but appear as 100.000 when they are rounded to three decimal places.

- The `print()` function, outer box. Formatted output, aligned in columns, is important for readability. We

are printing double values and we would like them rounded to three decimal places. So we write `<<fixed <<setprecision(3)`. These manipulators stay set until explicitly changed, so we write them once, outside the loop.

The loop is a for-each loop that defines `d` as the name of the current element. We write `<< d`. This code is equivalent to writing an iterator expression:

```
for (nIt = next.begin(); nIt != next.end(); ++nIt)
```

The width of an output field must be set separately for every field written out. Therefore, calls on `setw()` are written inside the loop, not before it.

- The `print()` function, inner box. If we printed the temperatures one per line, it would consume too many lines. So we use modular arithmetic to print five values per line in formatted fields. If the array subscript mod 5 equals 4, we know that we have printed five values on this line, so we print a newline character and enough spaces to indent the beginning of the next line.

## 17.5 What You Should Remember

### 17.5.1 Major Concepts

**Arrays and vectors of strings.** An earlier chapter introduced the ragged array of strings, which was initialized by string literals in that chapter. The data structure is even more versatile when implemented with C++ strings, which are dynamically allocated, because each string can contain data of any length.

### 17.5.2 Programming Style

**Use the proper arguments.** If you are writing a function to process a row of a matrix, pass as the argument a row of the matrix. Do not send the entire matrix as the parameter along with a row index to be used by the function in accessing the data. Write your functions to pass the appropriate amount of data, no more.

**Use type cast.** The `void*` pointer returned by `malloc()` and `calloc()` normally is stored in a pointer variable. While explicitly performing a type cast is not necessary, doing so can be a helpful form of documentation. Prior to the ANSI C standard, the explicit cast was necessary, and many programmers continue to use it out of habit.

**Use `free()` properly.** It is a good practice to recycle memory at the first possible time. Sometimes it is possible to reuse a memory block for different purposes before giving it back. This can improve program performance because the memory manager need not be involved as often.

**Check for allocation failure.** It is proper to check the pointer value returned by the allocation functions to make sure it is not `NULL`. If no memory is available, program termination is a logical course of action.

**Use `realloc()` properly.** Do not use `realloc()` too often, because it can reduce the performance of a program if data are copied frequently. Therefore, choose a size that is appropriately large but not too large. A common rule of thumb is to double the current allocation size. A final call to `realloc()` can be made to return excess capacity to the system when the dynamic data structure is complete and all data have been entered.

**Use the proper data structure.** It is important to choose which form of 2D structure you will use. Should it be defined at compile time or dynamically? Should it be a 2D matrix, an array of arrays, or an array of pointers to arrays? If the size is not known at compile time, dynamic memory is chosen. If the processing of rows is significant, one of the array structures is better suited. It is best to use a 2D matrix when the size can be defined at compile time and all the elements are treated equally.

**Use a setup function.** Use a setup function to organize data initialization and memory allocation statements. This improves code legibility and localizes much of the user interaction in one function.

**Use one buffer for string input.** It is a common practice to have one large buffer for string input, since the lengths of such input vary greatly. Each data item is read, then measured (using `strlen()`). An appropriate amount of memory is allocated dynamically for the input, then the input is copied into the new memory block and attached to some data structure. This frees the single long buffer for reuse.

### 17.5.3 Sticky Points and Common Errors

Dynamic memory allocation is a powerful technique but is prone to a variety of errors that cause programs to crash. The first four errors, below, all relate to overuse or underuse or misuse of the `free()` function. In avoiding one of these errors, it is important not to fall into another!

**Misuse of `free()`.** The two most common mistakes involving the use of `free()` are attempting to recycle a storage area that was not dynamically allocated and attempting to free a memory block that already has been recycled. Eventually, each of these is likely to result in an attempt to free storage that is part of some active object. The visible result may be garbage output or a sudden program termination. The cause may be very difficult to track down because it is not caused by the most recently executed part of the program. Further, the time and manner of crash will probably be different each time the program is run with different data.

**Memory leaks.** A memory leak occurs when the last pointer to a dynamic memory area is lost; the area was allocated but not freed, and remains a drain on the memory management system until the program terminates. If this happens repeatedly, and if the program runs for hours or days without terminating, system performance will be degraded. Thus, it is important to learn to free memory when it is no longer needed.

**Using a dangling pointer.** After a memory block has been freed, it never should be accessed again. When space is deallocated, it is logically “dead” but physically still there. If more than one variable is set to point at the area, a common error is to continue using the memory block. Once reassigned to a different portion of the program, the competing use eventually will corrupt the data.

**Using `realloc()`.** The dangling pointer problem also arises with regard to `realloc()`. If a new memory block is assigned, pointers into the old memory block become obsolete. Care must be taken to save the new memory address and use it to reset other pointers to elements within the data block.

**Uninitialized pointers.** It is easy to forget that every pointer needs a referent before it can be used. A pointer with no referent is like a pocket with a hole in the bottom; it looks normal from the outside but is not functional. A common error is to declare a pointer but forget to store in it the address of either a variable or a dynamically allocated memory block. This frequently results in a program crashing.

**Remembering array sizes.** Whenever an array is used, the actual number of data items in it must be remembered. Functions that process arrays must have two parameters, the array and the count. The count commonly is forgotten. Logically, these two items always should be grouped. In the next chapter, we use a structure that contains these two members so that we can pass the structure to functions as a single entity.

**Null character.** When allocating a memory block for an input string, remember to include space for the null character at the end.

### 17.5.4 New and Revisited Vocabulary

These are the most important terms and concepts discussed in this chapter.

<code>vector&lt;Type&gt;</code>	<code>front()</code>	<code>vector&lt;Type&gt;::iterator</code>
<code>push_back()</code>	<code>back()</code>	<code>++</code> on an iterator
<code>size()</code>	<code>begin()</code>	<code>*</code> on an iterator
<code>capacity()</code>	<code>end()</code>	<code>rename()</code> for a file
<code>subscript</code>	<code>find()</code>	<code>cin.ignore(1)</code>
<code>swap</code>	<code>sort()</code>	<code>cin &gt;&gt; ws</code>

## 17.6 Exercises

### 17.6.1 Self-Test Exercises

1. A basic step in many sorting programs is swapping two items in the same array. Write a function that has three parameters, an array of strings and two integer indexes. Within the function, compare the strings at the two specified positions and, if the first comes after the second in alphabetical order, swap the string pointers.

2. What standard header files must be included to use the following C++ features:

- (a) A vector
- (b) A string
- (c) sort() on a vector
- (d) ignore(1)

### 17.6.2 Using Pencil and Paper

1. What standard header files must be included to use the following C/C++ features:

- (a) new and delete
- (b) The ceil() function
- (c) A C++ string

2. In the heat flow program (Figure ??), we begin the simulation with an empty vector with size() = 0. Assume the initial capacity() = 16. Then we added values to the vector and it doubled every time the capacity was used up. Assume that we were monitoring node 9 and the simulation ran for 100 steps.

- (a) How many times did we double the array?
- (b) What was the final capacity of the array?
- (c) Altogether, how many `double` values were copied during the growing process?

3. Given these declarations, explain what (if anything) is wrong with the following allocation and deallocation commands:

```
double* p, *q;
int arr[5];
```

- (a) `p = new(double);`
- (b) `q = double[10];`
- (c) `delete q;`
- (d) `delete[] arr;`
- (e) `q = new double[10];`
- (f) `p = &q[2]; delete[] p;`

### 17.6.3 Using the Computer

1. Generating test data. Write a program that uses `srand()` and `rand()` to calculate a series of 1000 random floating point numbers. Write these to a file named “randfloats.in”.

2. Sorting a file of numbers. Modify the program from Figure 16.19 in three ways:

- Change it from interactive input and output to file input and output.
- Sort a file of numbers instead of a set of objects.
- Delete the two lines that prompt for and read the number of items to be sorted. Use a `vector` so that all of the numbers in the file can be stored in it.

Debug and test your program on a short file that contains ten numbers. Be sure that all ten inputs occur in the output file, in order. Then test your program on the long data file generated in the previous exercise.

3. Simulation of heat conduction.

Given the semi-infinite slab in Figure 17.8, determine the temperatures at nodes 1 through 30 inside the slab after time periods of 5, 10, 15, 20, 25, and 30 minutes have passed. Start with the functions in Figures ?? through 17.14 and modify them to print the results every 5 minutes and terminate after half an hour. Note that 30 minutes corresponds to  $p = 60$ .

4. A dictionary.

Make a dictionary data structure by reading in the contents of a user-specified text file one word at a time. Use C++ strings and store them in a vector. Sort these strings by using `vector::sort`. Print out the entries in your dictionary in alphabetical order. Do not display a word more than once. Instead, display a count of how many times each word appeared in the file.

5. A better dictionary.

Modify the program from the previous exercise. Do not display a word more than once. Instead, display a count of how many times each word appeared in the file.

6. String math.

Write a program to read any two integers as character strings (with no limit on the number of decimal digits). Hint: Read a number into a string that will expand when necessary. Add the two numbers digit by digit, using a third string to store the digits of the result. Print the input numbers and their sum. Hint: To convert an ASCII digit to a form that can be meaningfully added, subtract '0'. To convert a number 0...9 to an ASCII digit, add '0'.

7. Easier said than done.

Read in the contents of a file of real numbers for which the file length is not known ahead of time and could be large. Write the numbers to a new file in reverse order. Abort the program if the file is so long that the data cannot be held in the computer's memory.

8. Sparse matrix.

Many applications involve using a matrix of numbers in which most entries are 0. We say that such a matrix is *sparse*. Assume that we are writing a program that uses a sparse 100 by 100 matrix of `doubles` that is only 1% full. Thus, instead of 10,000 entries, there are only approximately 100 nonzero entries. However, the actual fullness varies, and there might be more than 100 nonzero entries. One representation of a sparse matrix is to store each nonzero entry as a structure of three members: the row subscript, the column subscript, and the value itself; that is, the value 20.15 in `matrix[15][71]` would be represented as the triple { 15, 71, 20.15 } . ,

- (a) Define a class named `triple` to implement the structure for one element of the sparse matrix.
- (b) Declare a class `Matrix` with a data member that is a vector. Use the vector to store all the triples in the matrix.
- (c) Write a function, `getMat()` in the `Matrix` class. Write a sentinel loop that will read data sets (a pair of subscripts and a matrix value) from the keyboard until a negative subscript is entered. Create `Triple` objects and push them into the vector.
- (d) Write a function, `showMat()`, that prints the matrix as a table containing three neat columns (two subscripts and a value).
- (e) Write a main program that tests these functions. Testing programs on huge amounts of data is impractical. During development, use only a few Triples. Call `getMat()` and `showMat()` to read and print the values of the matrix. You will need more than one data set to test this program.

9. Partially sorting the matrix.

Start with the `matrix` program in the previous problem. Add to it a function, `sortMat()`, that sorts the array elements in increasing order by the first subscript. If two elements have the same first subscript, sort them in increasing order by their second subscripts. If both subscripts are equal, display an error comment. To perform the sort, adapt the sorting program in Chapter 16, printing the matrix before and after sorting.



# Chapter 18

## Array Data Structures

Two-dimensional arrays commonly are used in applied physics, engineering, and mathematics to hold numerical data and represent two-dimensional (2D) physical objects. In this chapter, we explore several different array data structures, including the matrix, arrays of arrays, arrays of pointers to arrays, and arrays of strings. We explore applications of these compound arrays, the type definitions used to facilitate their construction, and two-dimensional array processing. We consider multi-dimensional arrays briefly.

### 18.1 Concepts

#### 18.1.1 Declarations and Memory Layout

Figure 18.1 shows a two-dimensional array, sometimes called a **matrix**, used to implement a 4-by-4 multiplication table. We declare such an array by writing an identifier followed by two integers in square brackets. The first (leftmost) number is the row dimension; the second is the column dimension. Visually, **rows** are horizontal cross sections of the matrix and **columns** are vertical cross sections.

We initialize a two-dimensional array with a set of values enclosed in nested curly brackets; each pair of inner brackets encloses the values for one *row*. The result can be viewed, conceptually, as a rectangular matrix, but physically, it is laid out in memory as a flat data structure, sequentially by row. Figure 18.2 shows two views of a 3-by-4 array of characters: The two-dimensional conceptual view and the linear physical view. Technically, we say it is stored in **row-major order**. All the slots in a given row will be adjacent in memory; the slots in a given column will be separated. This can have practical importance when dealing with large matrices: Row operations always will be efficient; column operations may not be as efficient because adjacent elements of the same column might be stored in different memory segments.

We refer to a single slot of a matrix using **double subscripts**: the row subscript first, followed by the column subscript. Each subscript must have its own set of square brackets. For example, to refer to the first and last slots of the matrix in Figure 18.1, we would write `mult_table[0][0]` and `mult_table[3][3]`.<sup>1</sup>

---

<sup>1</sup>Experienced programmers who are new to C must take care. They might be tempted to write `multTable[j,k]`, which is correct in FORTRAN. This means something obscure in C, and it will compile without errors. However, it does not mean the same thing as `multTable[j][k]`. (The comma will be interpreted as a comma operator, which is beyond the scope of this text.)

---

short multTable[4][4] = { {1, 2, 3, 4},	multTable	[0]	[1]	[2]	[3]
{2, 4, 6, 8},	[0]	1	2	3	4
{3, 6, 9, 12},	[1]	2	4	6	8
{4, 8, 12, 16}	[2]	3	6	9	12
}	[3]	4	8	12	16

Figure 18.1. A two-dimensional array and initializer.

The conceptual view of a 3-by-4 array of characters is shown on the left; the subscripts of each cell are written in the corner of the cell. Note that subscripts are used to refer to individual cells but are not actually stored in the cell. The actual storage layout of this array is shown on the right with the subscripts under each cell.

conceptual view:

0	0,0 a	0,1 b	0,2 c	0,3 d
1	1,0 e	1,1 f	1,2 g	1,3 h
2	2,0 j	2,1 k	2,2 m	2,3 n
	0	1	2	3

actual layout of array cells in memory:

a	b	c	d	e	f	g	h	j	k	m	n
0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3

Figure 18.2. Layout of an array in memory.

### 18.1.2 Using `typedef` for Two-Dimensional Arrays

A programmer may create and use arrays, strings, and multidimensional arrays without defining any new type names. We never actually *need* to use a `typedef`, because it just creates an abbreviation or synonym for a type description. However, `typedef` often should be used with array types in place of the basic syntax, because `typedef` helps simplify the code and clarify thinking. More important, it often enables a programmer to work at a higher conceptual level and have less involvement with the actual implementation of a data structure. There are two conceptually different kinds of two-dimensional arrays, and the appropriate type definitions are quite different for the two varieties, although they are initialized and stored identically.

### 18.1.3 Ragged Arrays

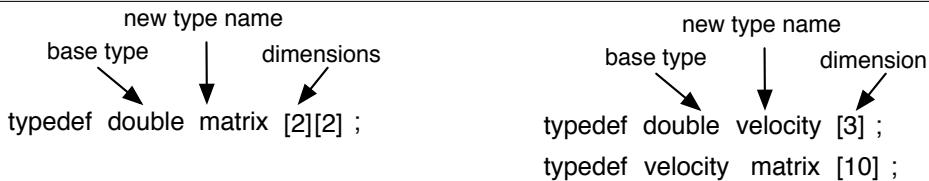
Another 2-D array data structure is the ragged array: an array (dynamic or declared) that points to C-strings (dynamic or constant). A ragged array is a natural data structure for use with a menu display. It is also the data structure used by C and C++ systems to pass command-line arguments to a main function. (This application will be covered in Chapter 20.)

In this section we introduce `menu()`, a function that is part of the tools library. It uses a ragged array to represent the menu selections. This array is diagrammed as part of the next demo program, Figure 18.5.

#### Notes on Figure 18.4: The menu function from tools.

##### *First box: Declarations in tools.hpp.*

- C++ makes it easy to use C++ strings, but there are often incompatibilities with non-constant C strings. When working with type `char*`, we often find ourselves adding `const` to the type. So we introduced a type name, `ccstring`, for constant-c-string, to make it more convenient to use the simple C-strings for simple applications.
- The menu function shown below is one of two in the tools library. The other is menus where the selection code is a character, not a digit.
- Parameters to `menu()` are a title, the number of possible choices, and a ragged array of item descriptions to display.

Figure 18.3. Using `typedef` for 2D arrays.

---

These declarations are in tools.hpp.

```
typedef const char* ccstring;
int menu( ccstring title, int n, ccstring menu[] );
```

This code is in tools.cpp.

```
// Display a menu then read and validate a numeric menu choice. Handle errors.
int
menu( ccstring title, int n, ccstring menu[] ) {
    int choice;
    for(;;) {
        cout <<'`n' <<title <<`n';
        for( int k=0; k<n; ++k ) cout <<"`t " <<k <<". " << menu[k] <<endl;
        cout <<" Enter number of desired item: `n";
        cin >> choice;
        if ( !cin.good() ) {
            cin.clear(); // Reset stream state to good.
            cin.ignore(1); // Clean garbage out of input buffer.
            choice = -1; // Set invalid choice to prevent loop exit.
        }
        if ( choice >= 0 && choice < n) break;
        cout << " Illegal choice or input error; try again. `n";
    }
    return choice;
}
```

---

**Figure 18.4. A menu function.**

***Second box: Displaying the menu.***

- The menu is an array with  $n$  menu selections. An ordinary for loop is used to display the selections from slot 0 to  $< n$ , so the menu will appear with selections numbered from 0 to  $n - 1$ . The program using this function must expect to get a 0-based subscript as the return value.
- The prompt for the menu choice makes clear that a numeric answer is expected.

***Third and fourth boxes: Reading a valid choice.***

- Reading the choice is easy. Deciding whether it is valid is not.
- The user could input a number that is too small or too large, or he could input a non-digit. We need to test for all three kinds of errors and produce an error comment if there is any kind of problem.
- If a letter is entered, it will cause the `>>` operation to fail. The fail flag in the stream will be set. At that point, nothing more can be done with the stream until the flag is cleared. Thus, to handle errors properly, the first thing to do is check the stream state and fix it if necessary.
- To prevent an infinite loop of the same error again and again, it is necessary to clear the offending character out of the stream. `ignore(1)` does this.
- Finally, to be sure that the error comment is printed, we set the choice to an invalid number.
- When control reaches the fourth box, the variable `choice` has a number in it and we can test whether that number is in the right range. If so, we break out of the validation loop. If not, the error comment is printed.

### 18.1.4 A Matrix

In one kind of 2D array, the data are a homogeneous, two-dimensional collection. Columns and rows have equal importance, and each data element has as close a relationship to its column neighbors as to its row neighbors. Programs that process this data structure typically have no functions to process a single row or column. Instead, they might process single elements, groups of contiguous elements, or the entire matrix. For example, consider an image-processing program in which each element of a matrix represents one pixel in a digital image. A **pixel**, which is short for “picture element,” is one dot in a rectangular grid of dots that, taken together, form a picture. That pixel has an equally strong relationship to its vertical and horizontal neighbors. Functions operate on entire images or on a rectangular subset of the elements, called a **processing window**. Rows and columns are equally important.

The general form of the `typedef` declaration for this kind of data structure is given on the left in Figure 18.3; note that the dimensions, not the type name, are written last. An example of its use for image processing is given in Section 18.3.

**Using a matrix.** The next program example illustrates the use of a 2D array for a practical purpose. Many road atlases have a table of travel times from each major city to other major cities, such as that in Figure 18.5. To use such a table, you find the row that represents the starting city and the column that represents the destination. The number in that row and that column is the time that it should take to drive from the first city to the second. We implement a miniature version of this matrix and use it to calculate the total driving time for a two-day trip, where you start in city 1, stay overnight in city 2, and end up in city 3. The program is in Figure 18.5.

**Notes on Figure 18.5. Travel time for a two-day trip.**

***At the top: Data diagrams.***

- The names of the cities covered by the table are defined as a ragged array in the second box.
- The table of travel times is a square matrix with one row and one column for each city. The value in each slot will be the number of minutes needed to drive from the row-index city to the column-index city.
- The integers `row` and `col` will be used in nested `for` loops to read the travel times from a file.

***First box: the environment.***

- This program uses the `menu()` function from the tools library.
- The number of towns is a constant used throughout the code. Such constants should always be defined at the top, not buried in the code.
- The file “minutes.in” contains the data shown in the diagram on the right.

***Second box: the matrix.***

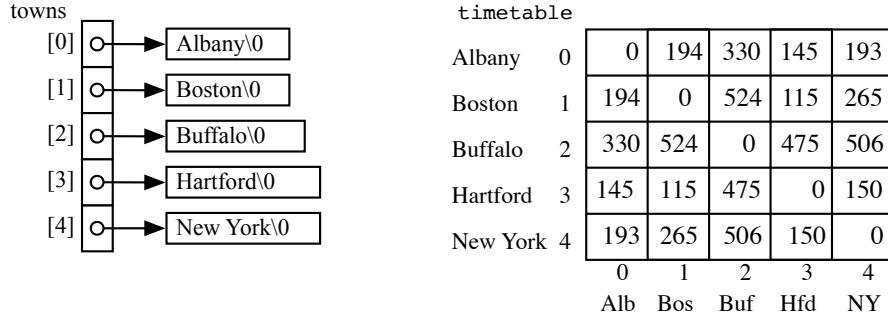
- The names of the cities covered by the table are defined as a ragged array (diagram at upper left) that will be used as a menu. The menu will be displayed three times to permit the user to select the source city (`city1`), layover city (`city2`), and destination city (`city3`). These cities will be used as subscripts for the travel-time matrix.
- Here we declare and allocate the matrix in the diagram at the upper right. We initialize it in the third box.

***Third box: reading the input file.***

- The nested `for` loops used here are a typical control structure for processing a matrix.
- Note that we use meaningful names (instead of `i` and `j`) for the row and column indices. This makes the code substantially easier to understand.
- We keep this example simple by omitting normal error checking. We assume that the data in the file are not damaged and that the file contains the correct number of data values. In a realistic application, error detection would be necessary.

---

These are the cities and the matrix of city-to-city travel times used in the travel-time program.



```
#include "tools.hpp"
#define NTOWNS 5
#define INFILe "minutes.in"

int main( void )
{
    ccstring towns[NTOWNS] = { "Albany", "Boston", "Buffalo", "Hartford", "New York" };
    int timetable[NTOWNS][NTOWNS];

    int city1, city2, city3; // Cities along route of trip.
    int time;
    cout <<"\n Travel Time \n";

    ifstream minutes( INFILe ); // Data for travel-time matrix
    if (!minutes.is_open()) fatal( " Cannot open " INFILe " for reading." );

    for (int row = 0; row < NTOWNS) // Read travel-time matrix.
        for (int col = 0; col < NTOWNS; ++col) {
            minutes >>timetable[row][col];
        }
}

city1 = menu( " Where will your trip start?", NTOWNS, towns );
city2 = menu( " Where will you stay overnight?", NTOWNS, towns );
city3 = menu( " What is your destination?", NTOWNS, towns );

time = timetable[city1][city2] + timetable[city2][city3];

cout <<"\n Travel time from "<<towns[city1] <<" to "<<towns[city2]
     <<" to "<<towns[city3] <<"\n\t will be "<<time <<" minutes.\n\n";
return 0;
}
```

---

Figure 18.5. Travel time for a two-day trip.

***Fourth box: choosing three cities.***

- These three calls on `menu()` permit the user to select three cities from the `towns` list. The menu function lets the user select a prompt, the number of menu items, and an array of strings that describe the choices.
- The return value from `menu()` is the subscript for the chosen city in the `towns` array. The city numbers will be used to access both the list of cities and the matrix of travel times.

***Fifth box: calculating and printing the travel time.***

- To access a time from the table we give two subscripts; the number of the source city (the row) and the number of the destination city (the column).
- The total time is the sum of the times on each of the two days. Sample output follows. Normal operation is shown in the left column, error handling on the right:

Travel Time

```
Where will your trip start?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: 0
```

```
Where will you stay overnight?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: 1
```

```
What is your destination?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: 4
```

```
Travel time from Albany to Boston
to New York will be 459 minutes.
```

Travel Time

```
Where will your trip start?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: 6
Illegal choice or input error; try again.
```

```
Where will your trip start?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: -1
Illegal choice or input error; try again.
```

```
Where will your trip start?
0. Albany
1. Boston
2. Buffalo
3. Hartford
4. New York
Enter number of desired item: w
Illegal choice or input error; try again.
```

**18.1.5 An Array of Arrays**

In the other kind of 2D array, an **array of arrays**, the data are a collection of rows, where each row has an independent meaning. The data elements in each row relate to each other but not to the corresponding elements of nearby rows. Programs that process this data structure typically have functions that process a single row. The general form of the `typedef` declaration for this data structure is given on the right in Figure 18.3.

For example, consider a program that makes weather predictions. One of its data structures might be an array of winds measured by weather stations in a series of locations. Each wind is represented by an array of three `double` values, which give the magnitude and direction in Cartesian coordinates ( $x, y, z$ ). In Figure 18.6, we use `typedef` to give the name `velocity` to this kind of array. The variable `wind` is an array of velocities, representing the winds in several locations. The first coordinate of each wind is related to the second and third; taken together, they specify one physical object. However, the first coordinate of one wind has little relationship to the first coordinate of the next wind. You would expect various functions in this program to have parameters of type `velocity`, as does the function `speed()` in Figure 18.7.

---

```

typedef double velocity[3] Type velocity is an array of 3 doubles.
double speed( velocity #/) Given velocity, calculate wind speed.

velocity calm = {0, 0, 0}; No wind.
velocity wind[5]; // The winds for 5 towns.

```

---

**Figure 18.6.** Declaring an array of arrays.

**Using an array of arrays.** The program in Figure 18.7 implements a sample wind array representing five locations. Altogether, it contains 15 `double` values, five locations with three coordinates each. In this example, `wind[0]` is the entire velocity array for Bradley Field and `wind[3][0]` is the *x* coordinate of the velocity at Sikorsky Airport.

Weather stations at five locations phone in their instrument readings daily to a central station running this program. When a weather station reports its data, the data are recorded in the wind table for that day. The program accepts a series of readings, then prints a table that summarizes the data and the wind speeds at the locations that have reported in so far.

#### Notes on Figure 18.7. Calculating wind speed.

**First box: the type declaration.** We declare a type to represent the velocity of one wind. We use this type to build a two-dimensional array (several winds with three components each) and to pass individual winds to the `speed()` function.

#### Second box: the calculation functions.

- We define a 1-line function for squaring a number because it will make our formulas more readable.
- One reason to use `typedef` is that a `typedef` name makes it easier and clearer to write correct function headers. The `speed()` function operates on velocity arrays: the use of the `typedef` name makes that clear.

#### Third box: the data structure.

- Three objects are declared here as parallel arrays. Together, they form a masked table of wind velocities for several weather locations, whose names are listed in a form that can be passed to `menu()`.
- The array of names has one extra item on the end to make it simple to end menu processing and finish the program.
- The mask array is initialized to `false` values (0) to indicate that, initially, no stations have called in their data. Recall that if an initializer is given that is too short for the array, all remaining array locations will be initialized to 0.

#### Fourth box: entering the data.

- The `name` and `mask` arrays are parallel to the `wind` array. Once a city is chosen, that city number is used to subscript all three. When the wind information for that city is entered, the corresponding mask is set to `true`. If the same city reported a second set of data, it simply would replace the first.
- Even though `wind` is declared as an array of arrays, not a matrix, a single velocity component is accessed using two subscripts.
- For simplicity, error checking is omitted here. In a realistic application, the numbers entered would be tested for being reasonable and an error recovery strategy would be implemented to recover from accidental input of nonnumeric data.

#### Fifth box: calculating one wind speed.

- The argument to the `speed()` function is a single wind velocity vector, not the whole array of winds, because that calculation involves only one velocity. By passing only the relevant row of the wind array, we simplify the code for `speed()`. Inside the function, we focus attention on that single wind and can access the individual velocity components using only one subscript.

*Sixth box and Figure 18.8: printing the wind speed table.*

- Sample output follows, with dashed lines replacing repetitions of the menu:

```
Wind Speed
0. Bradley
1. Bridgeport
2. Hamden MS
3. Sikorsky
4. Tweed
5. --finish--
Enter number of desired item: 3
```

This program creates a table of wind velocities at several weather stations, calculates the wind speeds, and prints a report. It calls the function in Figure 18.8.

```
#include "tools.hpp"
#include <cmath>
#define N 5
typedef double velocity[3];

void printTable( ccstring names[], bool mask[], velocity w[] );
double sqr( double x ) { return x * x; }
double speed( velocity v ){ return sqrt(sqr(v[0]) + sqr(v[1]) + sqr(v[2])); }

int main( void )
{
    int city;
    double windspeed;
    velocity wind[N];
    bool mask[N] = { false };           // Initialize all masks to false.
    ccstring names[N+1] = { "Bradley", "Bridgeport", "Hamden MS",
                           "Sikorsky", "Tweed", "--finish--" };

    cout <<"\n Wind Speed\n" ;
    for (;;) {
        city = menu( " Station reporting data:", N + 1, names );
        if (city == N) break; user selected "quit"

        cout <<" Enter 3 wind components for " <<names[city] <<": ";
        cin >>wind[city][0] >>wind[city][1] >>wind[city][2];
        mask[city] = true;

        windspeed = speed( wind[city] );
        cout <<"\t Wind speed is " <<windspeed <<".\n";
        printf( "\t Wind speed is %g.\n", windspeed );
    }
    printTable( names, mask, wind );
    return 0;
}
```

Figure 18.7. Calculating wind speed.

---

This function is called from Figure 18.7.

```
void
printTable( ccstring names[], bool mask[], velocity w[] )
{
    int k;
    cout <<"\n Wind Speeds at Reporting Weather Stations\n";
    for (k = 0; k < N; ++k) {
        if (!mask[k]) continue;
        cout <<" " <<left <<setw(15) <<names[k] <<right;
        cout <<fixed <<setprecision(2) <<"(" <<setw(7) <<w[k][0]
            <<setw(7) <<w[k][1] <<setw(7) <<w[k][2] <<" ) speed: "
            <<setw(7) <<speed(w[k]) <<"\n";
    }
}
```

---

**Figure 18.8. Printing the wind speed table.**

```
Enter 3 wind components for Sikorsky: 1.30      2.10     -1.10
Wind speed is 2.7037.
```

```
-----
Enter number of desired item: 4
Enter 3 wind components for Tweed: 1.50      2.00     0.00
Wind speed is 2.5.
```

```
-----
Enter number of desired item: 0
Enter 3 wind components for Bradley: 0.73      1.60     -2.10
Wind speed is 2.73914.
```

```
-----
Enter number of desired item: 5

Wind Speeds at Reporting Weather Stations
Bradley      ( 0.73   1.60  -2.10 ) speed:  2.74
Sikorsky     ( 1.30   2.10  -1.10 ) speed:  2.70
Tweed        ( 1.50   2.00   0.00 ) speed:  2.50
```

- The parameters to the `print_table()` function are the three parallel arrays that make up the wind table. We use the `names` array to print the locations and the `mask` array to avoid printing “garbage” values for weather stations that have not reported in.

### 18.1.6 Dynamic Matrix: An Array of Pointers

Dynamic allocation of memory is not limited to one-dimensional arrays. Since 2D arrays are stored in contiguous memory, they also can be allocated as a single large area, a **dynamic 2D array**. However, accessing a particular element using the normal, two-subscript notation is not possible. (A function to perform this operation is considered in an exercise on image processing.) Alternatively, a 2D object can be represented as an array of pointers to arrays, using dynamic allocation repeatedly to create each part of the structure.

The result is a **dynamic matrix** data structure in which the rows can be efficiently manipulated and swapped, as illustrated in Section 18.4. Elements of this matrix can be accessed using the matrix name and two subscripts, as if the entire matrix were allocated contiguously. This is ideal for applications where entire rows of the matrix are treated as units. For example, in Gaussian elimination, each row represents an equation, and entire equations are swapped as the solution progresses.

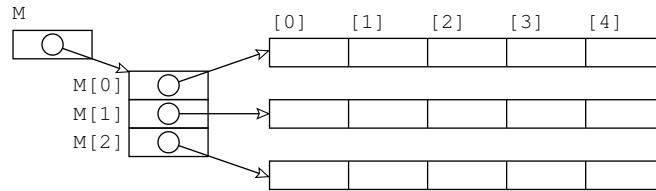


Figure 18.9. An array of dynamic arrays.

### 18.1.7 Multidimensional Arrays

Scientists and engineers use multidimensional arrays to model physical processes with multiple parameters. As in the two-dimensional case, a distinction should be made between multidimensional objects and arrays of matrices or matrices of arrays. C supports all these multidimensional data structures. Usage should be the guiding factor in deciding which to implement.

When using arrays of matrices or matrices of arrays, the rules for type compatibility of array parameters can become confusing. It is especially helpful to use `typedef` to define names for the subtypes, such as rows, and use those names to declare function parameters. The manipulation of uniform multidimensional structures can be more straightforward.

**Three-dimensional arrays.** The **dimensions** of a three-dimensional (3D) array usually are called *planes*, *rows*, and *columns*. The layout in memory is such that everything in plane 0 is stored first, followed by plane 1, and so forth. Within a plane, the slots are stored in the same order as for a two-dimensional matrix. Figure 18.10 shows a diagram of a 3D array with its subscripts.

When declaring a 3D array or a type name for a 3D type, each dimension must have its own square brackets, as shown in Figure 18.10. A 3D object may be referenced with zero, one, two, or three subscripts, depending on whether we need the entire object, one plane, one row of one plane, or a single element. For example, the middle plane in Figure 18.10 is `three_d[1]`. This plane is a valid two-dimensional array and could be an argument to a function that processes 2-by-4 matrices. Three-dimensional arrays are referenced analogously to matrices. For example, the last row of the last plane is `three_d[2][1]` and the last slot in that row is `three_d[2][1][3]`. A 3D function parameter is declared using a `typedef` name or with three sets of square brackets. Of these, the leftmost may be empty, but the other two must give fixed dimensions.

A `typedef` for a 3D array would extend the form for two dimensions, with the additional dimension, in square brackets, at the end. Multidimensional arrays may be initialized through nested loops or by properly structured initializers; a **3D initializer** would use sets of brackets nested three levels deep.

## 18.2 Application: Transformation of 2D Point Coordinates

An interesting application of two-dimensional arrays is the production of graphic images on the screen, often animated images. To move “objects” around on the screen, it may be necessary to rotate or translate them from their current position. This requires transforming the coordinates of the points constituting the object. In this section, we show a program that reads a set of point coordinates representing an object from a file and produces a new, transformed set of points. The code is written to transform a 2D image. However, by changing a `#define`, the same code will work with 3D images.

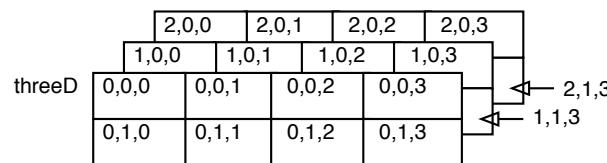


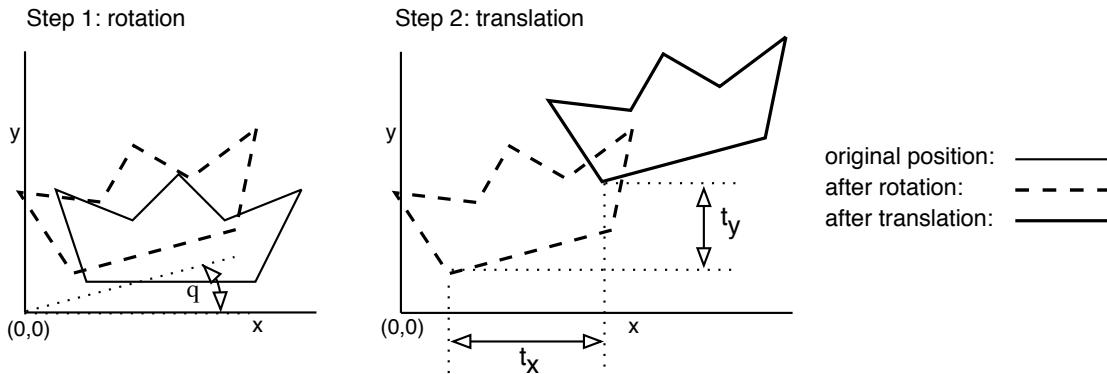
Figure 18.10. A three-dimensional array.

**Problem:** Write a program that reads in a set of points representing an object and produces another set of points based on a transformation (rotation and translation) of the original.

**Input:** (1) The name of a data file containing point coordinates, one point per line, each line containing the coordinates of one point, separated by a space. For a 2D drawing, there will be two coordinates,  $x$  and  $y$ , per point. For a 3D drawing, there will be also be a  $z$  coordinate.

(2) The transformation will be entered in the form of a counterclockwise (ccw) rotation angle,  $\theta$ , and numbers that are the translational changes in each of the 2 or 3 dimensions. To avoid confusion, the rest of this discussion is in the context of 2D drawings.

The diagram shows a rotation angle of 15 degrees and translation of (10, 6).



**Constant:**  $\pi$ .

**Formula:** The transformation usually is represented in symbolic matrix form as

$$p_{new} = R p_{old} + T$$

where  $p_{new}$  and  $p_{old}$  are given by  $(x, y)$  coordinate pairs,  $R$  is a 2-by-2 rotation matrix for an object rotating counterclockwise (left) or clockwise (right) through angle  $\theta$ :

$$R_{ccw} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad \text{or} \quad R_{cw} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$$

and  $T$  is a translation given by another  $(x, y)$  pair. Expanded, the matrix formula for counterclockwise rotation becomes two equations:

$$\begin{aligned} p_{new_x} &= p_{old_x} \cdot \cos \theta - p_{old_y} \cdot \sin \theta + T_x \\ p_{new_y} &= p_{old_x} \cdot \sin \theta + p_{old_y} \cdot \cos \theta + T_y \end{aligned}$$

The `sin()` and `cos()` functions are in the standard `math` library, which is included by `tools.hpp`.

**Output:** The original point coordinates are echoed to the screen in a table, and next to them are the new point coordinates that have resulted from the transformation. Display two decimal places.

**Limitations:** Since we are using vectors to store the points, there is no limitation on the number of points in a drawing.

**Figure 18.11. Problem specifications: 2D or 3D point transformation.**

This program is specified in Figure 18.11. It calls functions can be found in Figures 18.13 through 18.15.

```
#include "tools.hpp"                                // File: crown.cpp
#include "drawing.hpp"

int main( void )
{
    cout <<"Welcome to the " <<DIM <<"D drawing transformation program\n";

    Drawing dOld;                                     // Input the points of the drawing.
    Transform trans;

    Drawing dNew( dOld, trans );
    dOld.displayPoints( dNew );                      // print result table

    return 0;
}
```

**Figure 18.12. 2D point transformation—main program.**

A two-dimensional transformation is composed of two parts: a rotation and a translation. First, the object is **rotated** counterclockwise about the origin by an angle  $\theta$ . Then it is **translated**. This is a straight-line motion described by offsets in position along each of two orthogonal axes (usually the  $x$  and  $y$  axes). Given theta and the two offsets, an appropriate equation can be written for calculating the transformed coordinates of a point based on its original location. This situation is depicted in Figure 18.11 along with the specifications for the 2D transformation. The main program is listed in Figure 18.12, and the supporting functions are shown in Figures 18.13 through ??.

Classes are defined for a Drawing, a Point, and a Transformation. The first has only one data member: a vector. We define a class in order to have a place to put the relevant functions. The data members of the other two classes are arrays of doubles. We need to wrap these arrays inside classes to provide a place for functions and to make it possible to put the Points into a vector. The base type of a vector can be a primitive type or a class type, but it cannot be an array.

#### Notes on Figure 18.12: 2D point transformation—main program.

##### *First box: the environment.*

- We need the C math library, `<cmath>`, which is included by `tools.hpp`.
- The file `drawing.hpp` contains the class declarations for Drawing, Point, and Transform.

##### *Second box: Constructing the classes.*

- The Drawing constructor asks the user for a file that contains a drawing and reads it into memory. Errors during opening and reading are handled.
- The Transform constructor reads the rotation amount and translation amount from the keyboard and computes the transformation matrix.

##### *Third box: Using the transformation.*

- A second Drawing constructor initializes a drawing by transforming an existing drawing.
- Having computed the new drawing, both old and new are displayed, side by side, for comparison.
- This program does not write the new drawing to a file, but that might be useful.

#### Notes on Figure 18.13: Three class declarations.

##### *First box: Data members of Point.*

- Some real-life objects can be represented either by a structure or an array. In a previous chapter, we used a structure to represent a point in 2-space. Here we are using an array of doubles instead. There are several reasons:

- The processing done during a transformation is the same for both  $x$  and  $y$  components of a point.
- The code for that processing is easier to write and briefer using nested loops than using individual statements for each component.
- This code is written in such a way that it can be used for a 3D drawing as well as a 2D drawing. You can't do that if you must write the component name ( $x$  or  $y$  or  $z$ ) in the code instead of the component number (0...2).

**Second box: Function members of Point.**

- All of the Point functions are defined here, in the class declaration because they are all very brief. All three fit on one line.

---

```
#include "tools.hpp"                                // File: drawing.hpp
#define DIM 2                                         // For 2-dimensional drawings.
#define PI 3.1415927

//-----
class Point {
private:
    double pt[2];
public:
    Point() = default;
    Point(double x, double y){ pt[0]= x; pt[1]= y; }
    double& operator[]( int k ){ return pt[k]; }
    void print(){ cout <<" ( " <<pt[0] << ", " <<pt[1] << " )\n"; }
};

//-----
class Transform {
private:
    double rotation[DIM][DIM];
    Point translation;
public:
    Transform();
    void transformPoint( Point oldP, Point& newP );
};

//-----
class Drawing {
private:
    vector<Point> pts;                            // A drawing has many points
public:
    Drawing();
    Drawing( Drawing& oldD, Transform& trans );
    void displayPoints( Drawing& dNew );
    void getDrawing();
};

};
```

---

Figure 18.13. Three Class Declarations

- We need the default constructor to declare Point variables, such as the one in the Transform class.
- The constructor with parameters is needed to create a Point from the coordinates we calculate, prior to pushing the point into a vector that represents a new Drawing.
- The definition of `operator[]` is needed to enable the programmer to write code that looks like double subscripting on an array. The Point class delegates the subscript operation to its underlying array. The subscript operator returns a reference to a double so that the result of subscript can be used to store a double value.
- The print function is very ordinary. It was necessary during debugging, although it is not used anywhere in the finished application.

**Third box: Data members of Transform**

This program was originally written in C, with all the data members and functions thrown into one file. Translating it to a reasonable OO design was a challenge. The key came in realizing that Transformation needed to be a class on its own. It is the first example in this book of a `process` class. All other classes have been data or controller or container classes (like `vector`).

- A *process class* contains data members and functions needed to carry out a process. Instances of data classes are sent to it as parameters and returned by its functions.
- To perform a transformation, we need a 2-by-2 matrix of doubles for the rotation and an array of two doubles for the translation. These members define the transformation and really should not be part of any other class or of `main()`.

**Fourth box and Figure ??: the Transform functions.**

We imagine that this class could be part of a large graphical drawing package such as the one used to produce the drawings in this book. The prototypes

- The Transform constructor must get the data to define the transform. In this case, the data comes from the keyboard but it could come from a file or (more likely) from a GUI interface. The actual input data is used to compute the transformation matrix.
- There is only one function, and it carries out the process for which the class was designed. It applies the transformation to a point and returns another point. The algorithms come from linear algebra, where a *vector* is an array of numbers and a *matrix* is a rectangular array of numbers. Vectors are often used to represent points in 2-space or 3-space. Do not confuse this terminology with the C++ meaning of *vector*, which is a container class capable of storing an array of numbers.
- The transformation process starts by computing an algorithm called *dot product*, once for each dimension. The dot product of vectors *a* and *b* is:

$$a \cdot b = \sum_{k=0}^{DIM-1} a_k \times b_k$$

This is used twice to calculate the product of the rotation matrix and the old point, where *row<sub>0</sub>* and *row<sub>1</sub>* are the two rows of the translation matrix, considered as vectors.

$$newP = \begin{bmatrix} row_0 \cdot oldP \\ row_1 \cdot oldP \end{bmatrix}$$

- In Figure ??, the fifth box carries out the matrix operation. The outer loop calculates the two components of the new point. The inner loop does the dot product to calculate one new coordinate. Double subscripting is used to access the elements of the rotation matrix.
- The sixth box does the simpler operation of translating the figure, that is moving it up or down and right or left by adding or subtracting a shift amount from each coordinate.

**Fifth box of Figure 18.13: Data members of Drawing**

- This class is simply a wrapper for a vector of Points. We need a class instead of simply using a vector because we need a place to put the related functions.

*Sixth box of Figure 18.13 and Figure 18.15: Functions in the Drawing class.*

- There are two constructors. One is used to initialize the old Drawing from a file, the other initializes a new Drawing by transforming the old one.
- The code for handling the input stream is like all the other examples except for one feature: the Drawing class (not main()) prompts the user for the name of a file. This is a somewhat arbitrary decision. In this case, by opening and closing the file within the constructor, we avoid having a stream member of the class.
- The file handling follows the usual pattern of checking for all kinds of errors and handling errors by calling fatal. In this case, there is no reasonable way to recover from a missing or corrupted file, and little effort is required from the user to restart the program.
- For each line in the input file, two double values are read and used to construct a Point, which is then pushed into the vector. The final line of output was useful during debugging to prove that the vector was

---

These functions are called from Figure 18.12. The constructor inputs a rotation angle and a translation vector from which the 2D transformation is constructed, assuming a ccw rotation of the object. The function `transformPoint()` applies this transformation to a single passed point and returns new coordinates.

```

Transform:: Transform(){
    double theta;                                // rotation angle
    for (;;) {
        cout <<"Please enter counterclockwise rotation angle, in degrees: ";
        cin >>theta;
        if (theta >= 0 && theta <= 360) break;
        cout <<"Error: Angle must be in the range 0 - 360 degrees\n";
    }
    theta = PI * theta / 180.0;                  // convert angle to radians for math

    rotation[0][0] = cos( theta );      // compute rotation matrix elements
    rotation[0][1] = -sin( theta );
    rotation[1][0] = sin( theta );
    rotation[1][1] = cos( theta );

    cout <<"Please enter the translation amount ( X, Y ): ";
    cin >>translation[0] >>translation[1];
}

// -----
Point Transform:: transformPoint( Point oldP ) {
    Point newP; int r, c;                      // loop counters
    for (r = 0; r < DIM; ++r) {
        newP[r] = 0;                            // rotate by calculating the dot product
        for (c=0; c<DIM; ++c) newP[r] += rotation[r][c] * oldP[c];
    }

    newP[0] += translation[0];                // add translation vector to point
    newP[1] += translation[1];

    //oldP.print(); newP.print();           // debugging outputs
    return newP;
}

```

---

Figure 18.14. Functions for the Transform class.

being filled properly.

- The second constructor does all the work of the program, it transforms the points of the first drawing and pushes new points into the second drawing. As usual, the details of the transformation are hidden in the Transform class.
- The `displayPoints()` function is unusual because it deals simultaneously with two Drawings: the original is displayed on the left and the transformation on the right.
- This function illustrates an important fact about privacy in C++<sup>2</sup>: the parts of the implied parameter (the old point) and the parts of the explicit parameter (the new point) are both visible to this function. There is no need to use getter functions to access the private vector of either one.
- This code is also interesting because it uses double subscripts. There is a lot of design and a lot of machinery behind this simple-looking clause: `dNew.pnts[k][1]`. In this expression, the first subscript is interpreted by the vector class, the second one by the Point class.

**Results of transforming a point set.** The program was run on a data set defining the crownlike object originally shown in Figure 18.11. The object was rotated ccw by  $15^\circ$  and moved 6 units in the  $x$  direction and 10 units in the  $y$  direction. This would move the crown a little to the right and somewhat more upward. The output from the program follows. It is left to the reader to connect the dots.

```
Welcome to the 2D drawing transformation program
Please enter name of file containing object points: crown.in
The drawing has 7 points
Please enter counterclockwise rotation angle, in degrees: 13
Please enter the translation amount ( X, Y ): 2 6

Transformation of coordinates

Pt Old X Old Y New X New Y
1 2.00 8.00 2.15 14.24
2 6.00 7.00 6.27 14.17
3 10.00 9.00 9.72 17.02
4 14.00 7.00 14.07 15.97
5 18.00 8.00 17.74 17.84
6 16.00 2.00 17.14 11.55
7 4.00 2.00 5.45 8.85
```

## 18.3 Application: Image Processing

This example introduces several new programming techniques that are useful in a variety of applications:

1. Binary files.
2. Closing and reopening a file for a different use.
3. Reading and writing a huge file in one operation, using low-level I/O.
4. Allocating a dynamic array for a 2D image, and processing it using a subscript function to convert logical two-dimensional subscripts to physical one-dimensional subscripts.
5. The concepts of deep copy and shallow copy.
6. Skipping whitespace and comments in a file, as part of parsing an image header.

### 18.3.1 Digital images.

One task for which we use computers is processing digital images, for example, cropping them or restoring corrupted pictures. Consider the “snow” you see on the screen of a TV set with a poor antenna. Removing the snow, thereby producing a clearer picture, makes other image processing tasks easier. This image restoration can be accomplished using many different techniques, both simple and complex, with varying levels of success. The program we develop here performs a simple technique known as *image smoothing*. Snow is removed by introducing an overall blurring effect.

---

<sup>2</sup>This is unlike Java. In Java, one object cannot access the private parts of another object, even if they belong to the same class

---

These functions are called from Figure 18.12.

```

// -----
// Make a new drawing by reading points from an input file.
Drawing:: Drawing() {
    string fname;                                // name of data file
    double x, y;
    cout <<"Please enter name of file containing object points:  ";
    getline( cin, fname );
    ifstream fin( fname );
    if (!fin.is_open()) fatal( "Cannot open %s for input.", fname.data() );
    // Read data points until end of file or error occurs.
    for (int k = 0; ; k++ ) {
        fin >>x >>y;
        if (!fin.good()) {
            if (fin.eof()) break;           // End of file -- leave loop.
            else fatal( " Error reading file %s\n", fname.c_str() );
        }
        pts.push_back( Point(x, y) );
    }
    fin.close();
    cout <<" The drawing has " <<pts.size() <<" points\n";
}
// -----
// Make a new drawing by rotating and translating the pts of an old one.
Drawing:: Drawing( Drawing& oldD, Transform& trans ) {
    Point newP;
    for (Point p : oldD.pts) {
        newP = trans.transformPoint( p );
        pts.push_back( newP );
    }
}
// -----
void Drawing:: displayPoints( Drawing& dNew ) {
    cout <<"\n Transformation of coordinates\n" <<endl;
    cout <<"Pt Old X  Old Y      New X  New Y\n";
    for (int k = 0; k < pts.size(); k++) {    // put both points on one line
        cout <<fixed <<setprecision(2)
            <<setw(2) <<k+1 <<setw(7) <<pts[k][0] <<setw(7) <<pts[k][1]
            <<setw(10) <<dNew.pts[k][0] <<setw(7) <<dNew.pts[k][1] <<endl;
    }
}

```

---

Figure 18.15. Functions for the Drawing class.

Inside the computer, a picture must be stored in a discrete form. The format of a **digital image** typically is a header, containing information about the nature and size of the picture, followed by a grid of numbers, called pixels, where each number corresponds to the amount of light captured at that location by a camera. These numbers are typically scaled to a range of 0...255, where 0 corresponds to black, 255 is white, and the levels in between are shades of gray. This scaling is done so that each pixel can be stored in memory using only a single byte, a great memory savings when a typical image grid could be a square of size 1000-by-1000 numbers.

**Smoothing eliminates extreme pixel values.** Various situations (a dirty camera lens, dust inside the camera) can introduce speckles, called “snow” into a photograph. Usually we throw such damaged images away, but sometimes we prefer to rescue what we can of the photo.

The idea behind **smoothing** is that, in general, most pixel values are similar to those of their neighbors and the image intensity changes gradually across the picture. If one of the pixels in the image is corrupted, then its value probably has become quite a bit different than the values of the pixels surrounding it. Therefore, perhaps, a better value for the pixel would be based on its neighbor’s values. The simplest kind of smoothing calculation is to replace every pixel value by the average pixel value in a small square window centered about the pixel’s location. If a pixel is corrupted, this average should be much closer to the true value than the original value. If it is not corrupted, the true value will be changed slightly. The resulting image typically does not include the extremely erroneous pixel values, but some blurring of the picture does occur, especially for larger calculation windows.

Specifications for an image smoothing program are given in Figure 18.16. The various type declarations and functions that compose the program’s implementation are shown in Figures ??–??. The results of the smoothing operation are in Figure 18.25. The **fread()** and **fwrite()** functions are used in this application to read and write binary data files.

### 18.3.2 Smoothing an image.

**Notes on Figure 18.17: Main program for image smoothing.**

**First box: definition two global functions.**

- This main program is much like all earlier main programs: it sets up the environment, gets information from the user, opens a stream, instantiates the **Pgm** class, and calls the **Pgm** functions.
- It is different in two significant ways. First, the user interaction is extensive and has been moved out of **main** into a function. The prototype is here and the call is in the second box. The code is in Figure 18.18
- Second, a sophisticated technique is used by **main** to verify that the file name supplied by the user will not cause an unintentional file deletion. The function header is given here and the call is in the second box. The technique will be explained in the notes on Figure 18.18
- These two functions are written as global functions because they belong to **main()**, not to a class. They are declared to be **static** functions so that they *stay* inside the main module and are not visible to any other part of the program.

All information passed between the functions and **main()** must be passed through parameters. Note the ampersands in the first function prototype: they signify that the function will use these parameters to return information to **main()**. In the second function, the information is going from **main()** to the function, so no & is needed.

**Second box: User inputs.** **main()** is responsible for getting two file names and a mask size from the user. Two of these inputs will be validated.

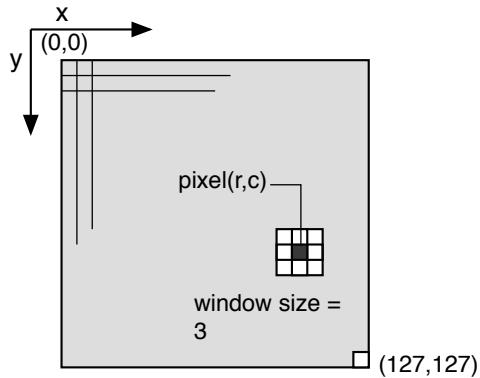
**Third box: Opening the output file.**

- A basic principle of program design is: *acquire all necessary resources before doing anything irreversible or heavily wasteful*. A corollary is: *open your output file before doing extensive processing or asking a human being to invest time in the application*.
- In this case, we want to be sure that an output file is available before beginning to input and process a potentially large input file. If we cannot open the output file, we abort now.

**Problem scope:** Write a program that will read in a digital image stored in P5 format and produce a new image in the same format by smoothing the pixel values of the original image.

**Input:** (1) The name of a data file that contains a digital image. (2) The name of a data file into which the new digital image will be stored. (3) The size of a processing window centered about each pixel, defining the neighborhood of values to be used in the averaging calculation.

The processing window follow:



**Constants:** The file format code is “P5”.

**Formula:** The average value of a processing window is the sum of the pixel values in the window divided by the number of pixels in the region. This formula does not hold for pixels around the border of the image, for which the window does not fit completely in the image. In such cases, the dimensions of the window must be cropped to confine the window within the bounds of the image.

**Output:** A new image is to be generated, with the same file header as the original image, but modified pixel values. Each new pixel value is the average value of the pixels in the window of the old image that is centered at the corresponding location.

**Limitations:** The input file must have the proper amount of data in it. The processing window size should be an odd number in the range 3 ... 11.

**Figure 18.16. Problem specifications: Image smoothing.**

**Fourth box: Creating the Pgm objects.**

- The object named `p` is the original photo that we wish to smooth, and `q` is the smoothed photo that we will calculate.
- The first function called here is the primary constructor for the `Pgm` class. It will initialize `p` by reading a photograph from the file whose name is stored in `inname`. The constructor reads in a brief header telling the dimensions and grayscale of the picture, then it allocates a dynamic array of the correct size and reads all the pixels into it.
- Before we can calculate the smoothed image, we must create a .pgm data structure and allocate memory for the pixels. The second function called here is a copy constructor. It initializes `q` by copying the non-dynamic parts of `p`. By doing so we make `q` the same size and grayscale as the original photo, but we do not copy the pixels. Then this copy constructor allocates a new array to hold the pixels that we will compute.

**Fifth box: Doing the work.** At this point, we have read in the original and prepared memory for the smoothed version. We can go ahead with the smoothing, then write out the result.

**Sixth box: cleanup.** Good programming style dictates that we free everything we allocate, as soon as we are done using it, so the file should be closed. We do this even when termination will be immediate.

**Notes on Figure 18.18: Getting parameters for the smoothing.**

This program is specified in Figure 18.16. It uses the parts in Figures 18.18 through 18.24.

```
#include "tools.hpp"
#include "pgm.hpp"                                // File: smooth.cpp

static void getParameters( string& inname, string& outname, int& maskSize );
static void checkOkToOpen( string outname );

int main( void )
{
    cout <<"This program smoothes a greyscale image\n";
    string inname;
    string outname;
    int maskSize;

    getParameters( inname, outname, maskSize );
    checkOkToOpen( outname );                      // abort if not ok.

    ofstream outFile( outname );
    if (!outFile.is_open()) fatal( "Cannot open %s for output", outname.data() );

    Pgm p( inname );
    Pgm q( p );

    q.smooth( p, maskSize );
    q.write( outname, outFile );

    outFile.close();
    return 0;
}
```

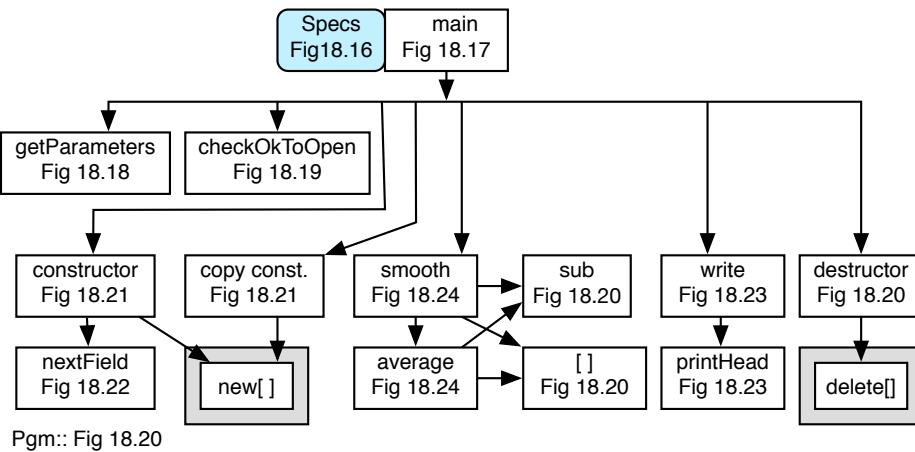


Figure 18.17. Image smoothing—main program.

---

```

void getParameters( string& inname, string& outname, int& maskSize )
{
    cout << "Please enter name of image file to open: ";
    getline( cin, inname );
    if (!cin.good()) fatal( "Error reading input file name" );
    cout << "Please enter name of file for result image: ";
    getline( cin, outname );
    if (!cin.good()) fatal( "Error reading output file name" );

    cout << "Enter size of square smoothing mask: ";
    cin >> maskSize;           // should be 3, 5, 7, 9 or 11
    if (!cin.good()) fatal( "Error reading size of mask." );
    if (maskSize % 2 == 0)
        fatal( "Error: mask size must be an odd number.\n" );
    if (maskSize < 3 || maskSize > 11)
        fatal( "Error: mask size must be in range 3 - 11.\n" );
}

```

---

**Figure 18.18.** Getting parameters for the smoothing.

**First box: the parameters.** All three parameters are passed by reference (`&`), so the parameter names will be the addresses of main's variables. (The call is in Box two of the main program.) In the boxes below, input is read into the parameter variables, which actually stores the input in main's variables. When the `getParameters()` function returns, that input will be known to `main()`.

**Second box: the file names.** To run this program, we need an input file and an output file. This code is the minimal amount of work to make an appropriate interface. The program would work if the first test were skipped, but the user-interface would be worse. If there is an error reading the first file name, a program should not ask the user to enter the second name.

**Third box: the smoothing mask.** Here, we want only one simple number, but need to deal with three ways that the input could fail.

- The read operation will fail if the user enters a non-numeric character.
  - The mask value could be out of the range permitted by the program.
  - The value could be even, which is not allowed.
- 

```

void checkOkToOpen( string outname ) {
    ifstream testFile( outname );
    if ( ! testFile.is_open() ) return;

    char ch;
    testFile.close();
    cout << "File " << outname << " exists. Do you want to overwrite it? [y/N] ";
    cin >> ch;
    if (tolower(ch) != 'y') fatal("Aborted");
}

```

---

**Figure 18.19.** Prevent accidental overwriting of existing file.

In any of these cases, we call `fatal` and expect the user to run the program again. Very little effort has been wasted.

#### Notes on Figure 18.19: Prevent accidental overwriting of existing file.

##### *First box: testing whether the file exists.*

- Suppose you wish to open a file named “`x.pgm`” for output. If a file by the same name already exists in the current active directory, writing `ofstream myOut("x.pgm")` will delete the existing file. Sometimes that is OK. Often it would be an unintended problem.
- Some applications actually *ask* the user whether he wishes to overwrite an existing file, and this is considered good practice. So we want to find out whether the output file already exists, and we want to do it in a way that works for any operating system.
- We do this by trying to open that output file name as an input file. If that effort fails, we are happy: the file does not exist. So we return from this function.

##### *Second box: the file exists – now what?*

- Close that input file! We do not really want it.
- But we have discovered a potential problem and need to ask the user what to do next. If he does want to overwrite the file, we are happy and just return.
- But if the user says that he does *not* want to overwrite the file, we call `fatal()` and abort.

#### Notes on Figure 18.21: the Pgm class.

*First box: the typedef for pixels.* We use a `typedef` to provide a short, meaningful name for a long, nonspecific type. We are using `char`s because the pixels are each one byte, and `unsigned` because a pixel value is in the range `0...255`.

##### *Second box: declaration of a window type.*

- The smoothing process will compute the new pixel value by averaging the pixel values in a square area surrounding each pixel of the image. We need this type to define the position of that averaging window as we move it across the image.
- We do not need to define a type for this purpose; we could use an array of four integers. The reason for declaring a type name and part names is to make the code easier to read and understand.
- This type is a struct, not a class, for simplicity and because there are no functions associated with it. It is a helper-type for the `Pgm` class. All of its members are public.

*Third box: data members.* The list of data members includes all required fields of the `Pgm` format plus one member for internal use.

- A `Pgm` file in `P5` format has these fields, separated by whitespace:
  - The characters “`P5`” must come first. We reserve a 3-char array for this code to allow space for a null terminator. All three slots are initialized to null, and the first two will be changed by the constructor. The null in the third slot allows us to print the 2-character code as a string.
  - The width and height of the image, in pixels, come next.
  - The gray scale value tells us how many different pixel values are allowed in the photo. Normally, this is `255`, but it could be a more restricted number.
  - At least one whitespace (probably a newline) must follow the `maxGray`.
  - After that are the pixels of the image.
- There should be  $length \times width$  pixels. If there are too many or too few, it is an error. This number is computed here and stored in a data member because it is used by several of the functions in this class.
- Comments are allowed anywhere in the header and are likely to come after the “`P5`”. A comment starts with a `#` character and ends at the next newline.

***Fourth box: private functions.***

- These two functions are private because they are not intended for use outside the Pgm class.
- These functions are both *const functions*: *in a function prototype, the keyword const just before the semi-colon declares that the function does not modify the data members of the class.*
- *The nextField() function is called several times from the Pgm constructor to help parse the header of the Pgm file.*
- *The average() function is called from smooth to perform the calculation on which smoothing is based.*

***Fifth box of Figure 18.20: constructors and a destructor.*** Here we give an overview of the three methods. See Figure 18.21 for more detail.

- The primary constructor for this class opens the image file, reads in the header, allocates dynamic memory, and then reads in the pixels. Errors due to incomplete files and inconsistent files are handled.
- The second constructor is a copy constructor<sup>3</sup> that defines how to initialize a new class instance by copying an existing one. Every class automatically has a copy constructor that copies all the bits of the original

<sup>3</sup>A constructor whose parameter is type `const classname&` is always a copy constructor.

---

```

#pragma once                                // File: pgm.hpp
#include "tools.hpp"
typedef unsigned char pixel;

// A rectangular window, given by its upper left and lower right corners
struct Window { int left, top, right, bot; };

class Pgm {
private:
    char filetype[3] = {'\0'};           // 2 characters designate the file format.
    int width;
    int height;
    int maxGray;
    int len;                           // length of pixel array = width * height
    pixel* image;

    void nextField( istream& s ) const;
    pixel average( Window w ) const;

public:
    Pgm( string inname );
    Pgm( const Pgm& oldP );
    ~Pgm() { delete image; }

    pixel& operator[]( int k ){ return image[k]; }
    int sub( int row, int col ) const { return row * width + col; }

    void smooth( const Pgm& p, int maskSize );
    void printHead( ostream& out ) const;
    void write( string outname, ofstream& outFile );
};


```

Figure 18.20. The Pgm class.

object into the new one. However, this default definition can be changed, and that is what we are doing here.

- The class destructor is the third line. We need an explicit destructor in this class because the constructor uses dynamic memory. It allocated a new pixel array named `image`, and we delete that array here.

**Sixth box: subscript functions.** These two functions are both inline. When an inline is called, the entire body of the function is written in the caller's code in place of the function call. For one-line functions, this increases efficiency.

- The first line defines how to do a subscript operation on a `Pgm` object. The function body delegates that subscript operation to the `image` inside the `Pgm` object. It returns the address (`&`) of the selected array slot, which can then be used on either side of an assignment statement.
- The syntax that you see in this function declaration is used only to define new methods for subscript. Calls on subscript use the familiar syntax for subscripting. For example, the last line of the `smooth` function calls `operator[]`.
- The second function, `sub` translates a logical 2D subscript to a physical 1D subscript. This is necessary because dynamically allocated arrays are all 1D arrays. The programming student should learn this computation and recognize it when it appears.

**Seventh box: smoothing and output.**

- The `smooth` function performs the main task of this application. It is in Figure 18.24.
- The `write()` function outputs the completed, smoothed image to the previously-opened output file. It is in Figure 18.23.
- The `printHead()` function is called while writing the modified image to a file. It was written as a separate function because it was also very useful to monitor the program's actions during debugging.

**Notes on Figure 18.21: The primary `Pgm` constructor.** This is an unusually complex function for reading the contents of a file because a `Pgm` file is written partly as a text file and primarily as a binary file. On an OS platform that distinguishes between text and binary modes, the file must be opened, closed, opened again, and closed again. Both the open operation and the read operation could generate errors that must be checked. So a seemingly simple job takes a whole page of code.

**First box: reading the file header.**

- The `Pgm` header is written in text mode. It must start with the 2-letter code “P5”. Reading these chars one at a time, as chars, seems to be the easiest and safest way to do the input. The 3rd slot in the char array has previously been initialized to a ‘\0’ so that we can print out the code as a string.
- Next, we expect to find three integers. They might be separated by a single whitespace character, but there could also be several spaces or an entire comment to skip. This is enough complexity to write a function to skip over this irrelevant material. See Figure 18.22. We alternate reading a number and skipping whitespace.
- Finally, we read the single required whitespace char at the end.

**Second box: checking for and handling errors.** Note: if any kind of error happens while reading a file, nothing in the file itself or in the stream will change until the stream error flags are cleared.

- It is not necessary to check for errors after every read operation because any error, anywhere in the header, makes the file invalid. We can attempt to read the whole header, and check at the end to be sure the stream is still “good”.
- The two `if` statements in this box check for all error that might have happened while reading this file header. Anything bad causes immediate termination.

**Third box: calculation and output.** When we get here, we have finished reading the part of the file that is written in text mode. Only the binary portion follows.

- We need to change mode from a text file to a binary file. To do that, we must close the file and reopen it using `ios::in | ios::binary`.

```

Pgm:: Pgm( string inname ) {                                // Initialize a pgm object from a file.
    int n, ch;
    // Open file the first time to read in text mode.
    ifstream pgFile( inname );
    if (!pgFile.is_open()) fatal( "Cannot open input file %s\n", inname.data() );

    // Scan and parse file header -----
    pgFile >> filetype[0] >> filetype[1];      // Get file type
    nextField( pgFile );                         // skip to start of width field
    pgFile >> width;                           // skip to start of height field
    nextField( pgFile );                         // skip to start of maxgray field
    pgFile >> height;
    nextField( pgFile );
    pgFile >> maxGray;
    ch = pgFile.get();                          / Required whitespace character preceding pixels.

    if (!pgFile.good() || !isspace(ch) || strcmp( "P5", filetype ) != 0)
        fatal( "%s is not a PGM file.", inname.data() );
    if (maxGray<1 || maxGray > 255)           // Check range of maxgray
        fatal( "MaxGray must be between 1 and 255" );

    // Remember current position in the file for seeking back to it later.
    streampos filePos = pgFile.tellg();
    if (filePos<0) fatal("Can't get file pointer after reading header");
    pgFile.close();
    pgFile.open(inname, ios::in | ios::binary); // Reopen file in binary
    pgFile.seekg(filePos); // Restore file pointer.
    if (!pgFile.good()) fatal("Can't restore file pointer");

    image = new pixel[len];                  // Allocate storage for pixel data.
    pgFile.read( (char*)image, len );        // Get pixel data.
    if (!pgFile.good()) fatal("Incomplete pgm file '%s'", inname.data());

    // Check for eof -----
    pgFile.get();
    if (!pgFile.eof()) fatal("File '%s' contains unread data", inname.data());
    pgFile.close();
}

// -----
// Copies an existing Pgm image into this Pgm object.
Pgm:: Pgm( const Pgm& old ) {
    this = old;                            // Shallow copy.
    image = new pixel[ len ];              // Allocate new pixel array
}

```

Figure 18.21. The Pgm constructors.

---

```

// Skip past inter-field whitespace and comments in pgm header. -----
void Pgm:: nextField( istream& s ) {
    char c;
    string junk;
    for(;;) {
        s >> c;
        if (s.eof()) return;
        if (c != '#') break;
        getline( s, junk );
    }
    s.unget();
}

```

---

**Figure 18.22.** Parsing and printing the header.

- But after reopening, we need to get back to exactly the current position in the file. So before closing, we need to find out and remember where we are. That is what `tellg()` does: it tells us the current position of the file pointer, returned as a value of type `streampos`. Of course, we check for errors. That is tedious but important.
- After executing `tellg()`, `close()`, then `open()` and `seekg()`, we are back at the end of the file header and ready to read in binary.

***Fourth box: allocation.***

- To prepare for reading the pixels, we allocate an array long enough to hold `length * width` pixels.
- The function `read()` is a low-level input operation that will read a specified number of bytes into a given array. Unfortunately, it has methods for reading and writing type `char`, but no methods for type `unsigned char`s. We “work around” this limitation by casting the image array pointer to type `char*` for the read operation, and again for the write operation later. This works without problems.
- If an eof occurs during the read operation, there are not enough pixels in the file to satisfy the read request. This could mean that the Pgm header is corrupted or that the end of the file has been removed. In either case, there is no point in going on, so we abort.

**Notes on Figure 18.21, last box: The Pgm copy constructor.** Every class has a copy constructor, defined by default. It copies the bits from one object to another. However, if the first object has dynamic allocation attached, only the pointer is copied, not the entire information. The result is a recipe for trouble: two objects end up pointing at the same dynamic allocation.

***The Pgm copy constructor.***

- In main, we allocate two Pgm objects, `p` and `q`. `p` is initialized by the primary Pgm constructor reading an image from a file. `q` is created to hold the modified image. The modified image should be the same size and shape as the original, so the header information for `q` is the same as for `p`. However, the two objects must end up pointing at different pixel arrays.
- The first line of this method copies the entire old object into the new one, including the image pointer. The second line computes the total size of the dynamic area.
- The second line allocates new memory and stores it in the new image. `q` is now a complete Pgm object, independent of `p`, and ready to use to store the modified pixels.

**Notes on Figure 18.22. Parsing the header.** The function `nextField()` is a private helper function for the Pgm constructor. Its purpose is to skip comments and whitespace embedded in the Pgm header. The use of `unget()` is new to this text.

---

```

// Print the pgm header. -----
void Pgm:: printHead( ostream& out ) const {
    out << filetype << "\n"
        << width << " " << height << "\n" << maxGray << endl;
}

// Print the entire Pgm image. -----
void Pgm:: write( string fileName, ofstream& pgmFile ) {
    printHead( pgmFile );                                // Write .pgm header data
    pgmFile.close();

    pgmFile.open( fileName, ios::out | ios::app | ios::binary );
    pgmFile.write((char*)image, len);                     // Write pixel data.
    if (!pgmFile.good()) fatal("Error writing pgm file '%s'", fileName.data());
    pgmFile.close();
}

```

---

**Figure 18.23.** Writing the Pgm file.

**First box: the skipping loop.**

- There is no simpler way to eliminate whitespace and comments than to read the file one byte at a time and test each byte. Remember that the operator `>>` skips leading whitespace, which is part of the goal in this application.
- In a Pgm file, comments extend from the `#` to the next newline character. We read the header, one character at a time, checking for the `#`. When we find it, we skip the rest of the comment line.
- A non-whitespace character that is not a `#` causes us to leave the loop. At that point, we have read the first digit of one of the three numbers that describe the image. We have gone too far!

**Second box: Put it back!** This is a common situation, and C++ provides a function to handle it. The call on `s.unget()` moves the stream cursor back one character, effectively putting the most recent character read back into the stream. Now the stream cursor is positioned properly to read the next number. Note: you can only “unget” one character.

**Notes on Figure 18.23: Writing the Pgm file.** This is considerably simpler than reading a Pgm file.

**First box: Writing the header.** This code is straightforward. It is simpler than reading the header because no parsing is necessary. We supply spaces and newlines where they are needed.

**Second box: Writing the entire image file.** We need to reverse the process of reading the file. However, again, there are no unknowns and the code is much much simpler.

- The output file was opened in text mode in the Pgm constructor. After printing the header, we need to close the stream and reopen it in binary mode.
- When we reopen the output file in binary mode, we also specify append mode: the pixels will be written at the end of the header information that is already in the file.
- `write()` is a low-level output command that writes a specified number of bytes to a file. As with `read()`, we need to cast the pixel array from type `unsigned char*` to type `char*` before `write` can use it. This works without problems.

**Notes on Figure 18.24. The smoothing calculation.** The smoothing calculation computes each pixel of the new image by averaging the values of its neighbors in the old image.

***First box (outer): Processing one pixel.***

- The smoothing process involves nested loops, where the inner loop processes all the columns in one row of the image and the outer loop processes all the rows of the image. The nested loop structure in this function follows the typical form for processing a 2D array.
- Each pixel must be processed independently of the others. It is necessary to save the results of processing each pixel into a new image; otherwise, the processing of successive pixels would incorporate both old and new pixel values, which would give a distorted result.

***First inner box: determining window bounds.***

- The processing window is computed separately for each pixel. It is centered about the pixel in question and extends `maskSize/2` pixels in each direction from that center. Integer division by 2 makes sure that integer limits are generated.
- Around the borders of the image, portions of the window may extend past the image boundary. Since there are no pixels in these areas to use, the window bounds must be restricted to stop at the actual edges of the image. For example, if the proposed processing window would have a negative subscript on the left side, it is set to 0 instead.

***Second inner box: calculating the new pixel value.***


---

```

void Pgm:: smooth( const Pgm& oldP, int maskSize )
{
    const int offset = maskSize/2;           // offset from center to edge of window

    for (int r = 0; r < width; r++)
        for (int c = 0; c < height; c++) {
            Window mask;                   // boundary limits of processing mask
            mask.left = max( 0, c-offset );
            mask.top = max( 0, r-offset );
            mask.right = min( c+offset, width-1 );
            mask.bot = min( r+offset, height-1 );

            // compute and store new pixel value -----
            image[ sub(r, c) ] = oldP.average( mask );
        }

    // -----
    // Compute average value of pixels in bounded window of image.
    //

    pixel Pgm:: average( Window mask ) const
    {
        int sum = 0, count = 0;           // sum and pixel-count for average
        for (int r = mask.top; r <= mask.bot; r++)
            for (int c = mask.left; c <= mask.right; c++) {
                sum += image[ sub(r, c) ];
                count++;
            }
        return sum / count;
    }
}

```

---

Figure 18.24. Smoothing.



Figure 18.25. Results of image smoothing program.

- The original image and the bounds of the processing window are passed to the `average()` function to do the actual calculation. By putting the calculation in a separate function, we keep the process of cycling through all the pixels separate from the process of calculating the value of one pixel. We also avoid writing a nest of `for` loops that is four levels deep. Both of these goals are worthy design goals.
- The single line of code in this box calls three of the functions we have defined. First, the `average()` function is called to compute the new pixel value. Then the row and column numbers of the current pixel sent to `sub()` and converted to a 1-D subscript. Finally, that subscript is used by `operator[]` to access `image`, the dynamic pixel array.

*Last box: the double loop in the average() function.*

- This is a straightforward summing loop to add up the pixel values in a square area. The `Window` object contains the beginning and ending subscripts for the rows (outer loop) and for the columns (inner loop).
- When the loop ends, all  $n^2$  values have been summed. The last line computes and returns the average.

**Results of smoothing an image.** The program was run on a sample image. The results are shown in Figure 18.25. The original image on the left was corrupted by a small amount of “snow.” A window size of 3 was chosen for processing, which produced the image on the right. As can be seen, the extreme white values have been removed, but the image has been blurred. Other image restoration techniques, which are more complex, can remove the corrupted values without the blurring.

## 18.4 Application: Gaussian Elimination

**Gaussian elimination** is an algorithm for solving a system of  $m$  linear equations in  $m$  unknowns, as shown next. The goal is to find values for the variables  $a, b, \dots, m$  that simultaneously satisfy all the equations

$$\begin{aligned} c_{1,1}a + c_{1,2}b + \cdots + c_{1,m}m &= X_1 \\ c_{2,1}a + c_{2,2}b + \cdots + c_{2,m}m &= X_2 \\ &\vdots \\ c_{m,1}a + c_{m,2}b + \cdots + c_{m,m}m &= X_m \end{aligned}$$

We can use an  $M$  by  $M + 1$  matrix to contain the coefficients of the variables in the  $M$  equations, one row for each equation, one column for each unknown, and a final column for the constant term. To find the set of variable values that satisfy the system, we apply to the coefficient matrix a series of arithmetic operations that are valid for systems of equations. These operations include

- *Swapping.* Since the order of the equations in the matrix is arbitrary, we can exchange the positions of any two equations without changing the system. (But we cannot swap two columns because each column position is associated with a particular variable.)
- *Scaling.* Both sides of an equation may be divided or multiplied by the same number. That is, if we divide or multiply all of the coefficients and the constant term of an equation by a single number, the meaning of the equation remains unchanged.
- *Subtraction.* We can subtract one equation  $E_1$  in the system from another,  $E_2$ , and replace  $E_2$  by the result, without changing the constraints on the variable values that satisfy the system.

An algorithm that uses these operations, Gaussian elimination, can be used to compute the variable values that solve the **system of equations**.

We implement this algorithm by writing a function to perform each of the preceding operations, and a function, `solve()`, that uses them appropriately. The algorithm solves the system of equations in stages. At stage  $k$ , we select one equation, place it on line  $k$  of the matrix, then scale it by dividing all of the row's entries by its own  $k$ th coefficient. This process leaves a value of 1 in the  $k$ th column of the  $k$ th row of the matrix. The new  $k$ th equation then is used by the scaling and subtraction functions to reset the  $k$ th coefficient of every other equation to 0 while simultaneously adjusting the other coefficients. After  $M$  such elimination steps, the matrix (except for the last column) has a single element with the value 1 in each row and in each column. This corresponds to a set of equations of the following form, in which the last column of the matrix contains the solution to the system of equations:

$$\begin{aligned} 1 \cdot a &= X'_1 \\ 1 \cdot b &= X'_2 \\ &\dots \\ 1 \cdot m &= X'_m \end{aligned}$$

#### 18.4.1 An Implementation of Gauss's Algorithm

To solve a particular system of equations using this code, the user must first create a data file, `gauss.in`, that contains the data. The number of equations must be on the first line. Following that must be the coefficients of the equations, one equation per line. The constant term of each equation is the last entry on a line. The file we use in the example looks like this:

```
4
1   2   2   1   10
3  -1   0   5  -4
2   2   3   0   15
0   .5  0  -1    7  -9
```

The corresponding output, before solving the equations is:

```
Linear Equations to Solve:
-----
1.000 * a +  2.000 * b +  2.000 * c +  1.000 * d =  10.000
3.000 * a + -1.000 * b +  0.000 * c +  5.000 * d = -4.000
2.000 * a +  2.000 * b +  3.000 * c +  0.000 * d =  15.000
0.000 * a +  0.500 * b + -1.000 * c +  7.000 * d = -9.000
```

Use Gaussian elimination to solve  $m$  equations in  $m$  unknowns.

```
#include "gauss.hpp"

int main ( void )
{
    Gauss matrix;           // Array of equation pointers.
    matrix.print( cout, "\nLinear Equations to Solve:");

    bool solvable = matrix.solve();
    matrix.print( cout, "\n Equations after Elimination" );
    if (!solvable) fatal(" Equations inconsistent or not independent.\n");

    matrix.answers( cout );
    return 0;
}
```

Figure 18.26. Solving a system of linear equations.

**Notes on Figure 18.26. Solving a system of linear equations.** This is a typical OO main program. It instantiates `matrix`, an object of class `Gauss`, and displays the system of equations that will be solved. Then `main()` uses `matrix` to call functions that solve the system of equations and to print out the final solution. This is the right way to build a program: keep `main()` simple. The other functions, in the classes, do all the work.

The call chart in Figure 18.27 shows the structure of the Gaussian elimination program; the main program is in Figure 18.26, the classes in Figures 18.28 and 18.29, and the functions in Figures 18.30 through 18.36. A call chart is given in Figure 18.27; most standard system functions have been omitted from the chart.

#### Notes on Figure 18.28: The data structure.

- The Gaussian Elimination algorithm is based on operations on entire equations, where each equation is represented by one row of a matrix. Although we *could* use a two-dimensional array to hold the equation's coefficients, this data structure does not reflect the nature of the problem well: the program really is working with an array of equations, not a two-dimensional array of numbers.
- We wish to use dynamic allocation so we can solve a system of equations with any reasonable number of unknowns. A matrix that represents a system of  $n$  equations will have  $n$  rows and  $n + 1$  columns. Each column except the last represents one unknown. The last column represents the constant term in the

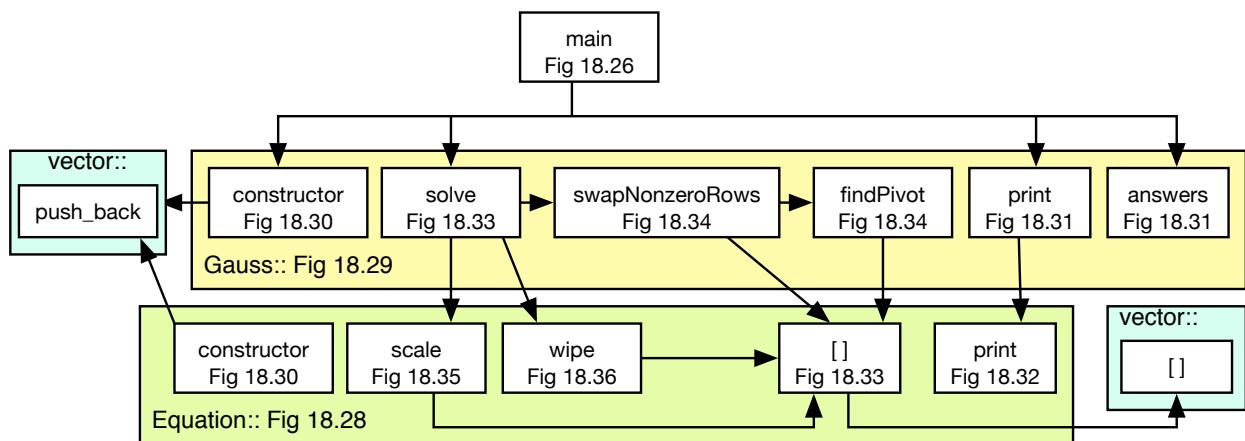


Figure 18.27. A call chart for Gaussian elimination.

This class is used by the Gauss class in Figure 18.29 to model one equation.

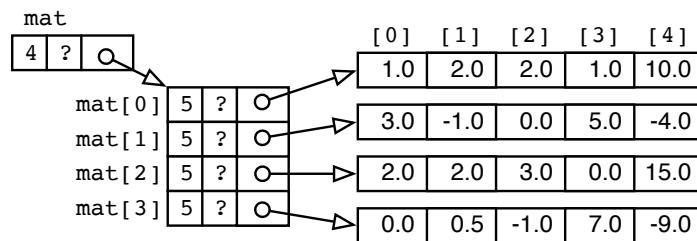
```
#pragma once
#include "tools.hpp"           // File:  equation.hpp
class Equation {
private:
    vector<double> coeff;
public:
    Equation( istream& eqIn, int nEq );
    void print( ostream& out );
    double& operator[]( int k ){ return coeff[k]; }

    void scale( int col );
    void wipe( Equation& piv, int pivRow );
};
}
```

Figure 18.28. The Equation class declarations.

equation.

- When the number of variables increases, both the number of rows and the number of columns increase. To allow any reasonable number of variables, we define the class **Equation** as a vector of **double** values and the class **Gauss** as a vector of equations. For the data file given earlier, the data structure will eventually have dynamically allocated parts that are related as shown in the diagram below. The question marks represent the data member in **vector** that tracks the actual length of the dynamic allocation. We simply do not know that length, and do not need to know.



Notes on Figure 18.28. The Equation class.

**First box: the data member.** The only data member of Equation is a vector. The class was created as a place to define functions that implement the row-operations of the elimination method.

#### **Second box: the usual suspects.**

There is nothing unusual here. These first two methods are what we expect to see in every class. The constructor prototype tells us that the coefficients for one equation will be read from an open stream. The definition of the subscript operator is expected: the outside world needs a way to access data stored in the coefficient vector. The operator definition is inline, the other two definitions are in Figure ??.

#### **Third box: the mathematics.**

The last two functions implement the row-operations that are part of the Gaussian Elimination algorithm. The Equation parameter is passed by reference (&) because it will be modified within the **wipe()** function. The definitions are in Figure 18.34.

Notes on Figure 18.29. The Gauss class.

---

This class is used by `main()` to model a system of equations.

```
#pragma once // File: guass.hpp
#include "equation.hpp"
#define EPS .0001 // Comparison tolerance for zero test.

class Gauss {
private:
    vector< Equation > mat;
    int nEq;

public:
    Gauss();
    ~Gauss() = default;

    void print( ostream& out, ccstring message );
    void answers( ostream& out );

    bool solve(); // k is the column subscript
    int findPivot( int k );
    bool swapNonzeroRow( int k );
};

};
```

---

**Figure 18.29.** The Gauss class declaration.

**First box: the constant.** We need a comparison tolerance in certain portions of the algorithm to test whether coefficients equal (or nearly equal) zero. The `find_pivot()` function uses this constant to identify which equations have a nonzero coefficient in the current column. The tolerance is needed to avoid floating-point overflow caused by division by 0 or a nearly 0 value. The constant `EPS` is defined here, rather than in the function that uses it, to make it easy to find and change if the user requires more or less precision.

**Second box: the data members.**

- This program is able to handle a system of equations with any reasonable number of equations, `neq`. Since we do not know at compile time how many equations there will be, we use a vector to store them.
- The variable `neq` is the first thing read from the input file and determines how many lines of data (one equation per line) will be read. This number is used throughout the Gauss class. It is not necessary to store it as a class member because the same number will be stored inside the vector. However, the code is easier to write, read, and debug with this number stored in a convenient place.

**Third box: the constructor and destructor.**

- The class constructor is defined in Figure 18.30 and will be discussed in the notes for that Figure.
- It is not necessary to define a destructor for this class because all the dynamic allocation is hidden inside the vectors, which manage it. If a class does not have a definition for a destructor, C++ will provide a do-nothing destructor for the class. However, it is considered good style to define a destructor even if it does nothing. The phrase `= default` declares that the destructor should do nothing.

**Fourth box: output.** These functions are defined in Figure 18.31.

- The `answers()` function is used once at the very end of the program to print the solution.
- The `print()` function is used just after the input phase is finished and just before the answers are printed. During debugging, it was used after every stage of the solution and was essential to track and correct the progress of the algorithm.

The Gauss constructor is called from `main()` in Figure 18.26; it opens the input file, reads equation data from it, and calls the Equation constructor.

```

Gauss:: Gauss() {
    // Open input file and read number of equations. -----
    ifstream eqInput( "gauss.in" );
    if ( ! eqInput.is_open() ) fatal ( " Cannot open gauss.in" );
    eqInput >>nEq;

    // Read in all the equations. -----
    for ( int k = 0; k < nEq; ++k ) // Read and install an equation.
        mat.push_back( Equation( eqInput, nEq ) );

    eqInput.close();
}

// -----
Equation:: Equation( istream& eqIn, int nEq ) {
    double temp;
    // Read coefficients for one equation. -----
    for ( int col=0; col<=nEq; ++col ) {
        eqIn >> temp;
        coeff.push_back( temp );
    }

    if ( !eqIn.good() ) fatal( " Read error or unexpected eof." );
}

```

**Figure 18.30. Constructing the model.**

#### *Fifth box: the mathematics.*

These function are defined in Figure 18.33. They will be discussed in the notes that follow that Figure.

#### Notes on Figure 18.29. Constructing the model.

**First box: the input stream.** The Gauss constructor opens and closes the input stream.

- The first line of the file determines how many other lines will be read.
- For each input line (one equation), Gauss calls the Equation constructor and passes it a reference to the open stream. The number of equations must also be a parameter because that determines how many coefficients will be read for this equation.
- When the Equation constructor returns, the result is immediately pushed into the vector of Equations.

**Second box: the input loop.** This loop reads all the coefficients and the constant term for one equation. As each number is read, it is pushed into the vector of coefficients.

#### *Third box: answers.*

- Whether we are printing the equations or the answers, formatting the output is important. Here we specify fixed point with three decimal places, right justified in 8 columns. (Right justification is the default.)
- The second line of the output statement uses a character-code “trick”. We want to print variable names a, b, c, d, etc. But we do not know how many letters we will need. So we calculate the ASCII code to print by adding the row number to the ASCII character ‘a’. This produces as much of the alphabet as needed. It is easy, fast, and portable to any system where the alphabet is given consecutive character codes.

---

These functions are called from `main()` in Figure 18.26 and used in various other places to provide debugging output.

```

// Print the matrix, formatted so that each row is an equation.
void Gauss:: print( ostream& out, ccstring message ) {
    out << message << endl;

    out << " -----\n";
    for (int row=0; row<nEq; ++row) {           // Print all equations.
        mat[row].print( out );
    }
    out << " -----\\n";
}

// -----
// Print each variable with its value (from last column of matrix).
void Gauss:: answers( ostream& out ) {
    for (int row=0; row < nEq; ++row) {
        out <<fixed <<setprecision(3) <<setw(8)
            <<" " <<(char)('a'+row) <<" = " <<mat[row][nEq] <<"\\n";
    }
    out << endl;
}

```

---

**Figure 18.31. Output functions for the Gauss class.**

**Third box: error handling.** Throughout this application, we are careful to do responsible error detection and print appropriate error comments. If any one of the equations in the system is unreadable, there is no way to solve the system and termination is appropriate.

To be OK, the *nEq* expected coefficients and the constant term must all be present. If any one is missing or causes a read error, we can detect the problem with a single test at the end of the read loop.

#### Notes on Figure 18.31: Output functions for the Gauss class.

**First box: the print function's message.** The `Gauss::print()` function was used during debugging to display the model after every row or column operation. To make sense of the display, it was important to know which operation had just been done. This problem was easily solved by including a message as a parameter to print; the messages (shown below) give the name of the function that just finished its work.

#### Second box: the outer print loop.

- A page full of columns of numbers is hard to read. We improve the situation a lot by printing a line of dashes above and below the system of equations, so that the many steps of the solution are clearly separated. The output below shows the first quarter of the solution process and the final answers. (The rest has been omitted for brevity.)
- The loop in `Gauss::print()` is repeated once for each equation in the system. It delegates the actual printing of the equation to the expert: `Equation::print()`. That method contains a loop to print all the coefficients.

#### Linear Equations to Solve

---

1.000 * a +	2.000 * b +	2.000 * c +	1.000 * d +	10.000
3.000 * a +	-1.000 * b +	0.000 * c +	5.000 * d +	-4.000
2.000 * a +	2.000 * b +	3.000 * c +	0.000 * d +	15.000
0.000 * a +	0.500 * b +	-1.000 * c +	7.000 * d +	-9.000

This function is called by Gauss::print() in Figure 18.31. It prints one row of the matrix formatted so that it looks like an equation.

```
// -----
void Equation:: print( ostream& out ) {
    int nEq = coeff.size()-1; // The vector has n coefficients and 1 constant.
    char op= '+';           // To print this between terms in output equation.
    for (int col=0; col < nEq; ++col ) {
        if (col == nEq) op = '=';
        out << fixed << setprecision(3) << setw(8)
        << coeff[col] <<" * " <<(char)('a'+col) <<" " << op <<" ";
    }
    out << setw(8) << coeff[nEq] <<"\n";
}
```

Figure 18.32. Output for the Equation class.

```
-----
Starting solve loop at pivRow 0
After swap.

3.000 * a + -1.000 * b + 0.000 * c + 5.000 * d + -4.000
1.000 * a + 2.000 * b + 2.000 * c + 1.000 * d + 10.000
2.000 * a + 2.000 * b + 3.000 * c + 0.000 * d + 15.000
0.000 * a + 0.500 * b + -1.000 * c + 7.000 * d + -9.000
-----

After scale.

1.000 * a + -0.333 * b + 0.000 * c + 1.667 * d + -1.333
1.000 * a + 2.000 * b + 2.000 * c + 1.000 * d + 10.000
2.000 * a + 2.000 * b + 3.000 * c + 0.000 * d + 15.000
0.000 * a + 0.500 * b + -1.000 * c + 7.000 * d + -9.000
-----

After wiping all rows.

1.000 * a + -0.333 * b + 0.000 * c + 1.667 * d + -1.333
0.000 * a + 2.333 * b + 2.000 * c + -0.667 * d + 11.333
0.000 * a + 2.667 * b + 3.000 * c + -3.333 * d + 17.667
0.000 * a + 0.500 * b + -1.000 * c + 7.000 * d + -9.000
-----

Starting solve loop at pivRow 1
. .
a = 1.000
b = 2.000
c = 3.000
d = -1.000
```

**Notes on Figure 18.32: Output functions for the Equation class.** The Equation::print() function prints one equation, with variable names and \*, + and = signs.

**Outer box: the loop.**

- The formatting is the same here as in Gauss::print().

This function performs the Gaussian elimination algorithm. It uses row operations to reduce the first  $M$  columns of the matrix to an identity matrix. At that point, the last column contains the solution to the problem. This function is called from Figure 18.26. It uses the functions in Figures 18.34 through 18.36.

```

bool Gauss:: solve() {
    for (int pivRow = 0; pivRow < nEq; ++pivRow) {
        cout << "Starting solve loop at pivRow " << pivRow << "\n";
        if (! swapNonzeroRow( pivRow )) // Do all rows have 0 in this column?
            return false;

        mat[pivRow].scale( pivRow );      // Make a 1 in mat[k][k].
        print( cout, "After scale." );   // For debugging.

        // Use the 1 in mat[k][k] to zero out the rest of column k.
        for (int row=0; row<nEq; ++row)
            if (pivRow != row) mat[row].wipe( mat[pivRow], pivRow );

        print( cout, "After wiping all rows." );
    }

    return true;
}

```

**Figure 18.33.** The `solve()` function.

- In a system of  $n$  equations, there are  $n$  unknowns. Each equation in the system starts with  $n$  coefficients, one for each unknown, and ends with a constant term. We want to print a `+` sign between every pair of coefficients and a `=` before the constant term. The char variable `op` is initially set to `'+'` and changed to `'+'` at the end of the loop.
- Currently, when a term is negative, the program prints both a `+` and a `-` sign in front of it. For even fancier output, we could test the sign of each coefficient and print either a `+` sign or a `-` sign, but not both.

**Inner box: the variable names.**

The actual variable names are computed here, as they were in `Gauss::print()`.

**Notes on Figure 18.33: The elimination algorithm.** The main loop of this function is executed once for each equation in the system. Its purpose is to manipulate the matrix so that, eventually, it has ones on the main diagonal (elements with the same row and column subscripts) and zeros elsewhere. To do this it calls three functions (in the three boxes).

**Outer box: To solve or not to solve, that is the question.** Each time around this loop one equation is selected from the set of remaining equations. It is called the *pivot equation* and is swapped into the first row of the matrix that has not already been processed.

**First inner box and Figure 18.34: Finding the next pivot row.** Remember that swapping the order of two equations in a system does not change the solution to that system. Here we swap rows and do row operations.

- On pass  $k$ , the `swapNonzeroRow()` function has two purposes:
  - To find the equation with the largest non-zero coefficient in column  $k$  and row  $\geq k$ .
  - To swap that equation with whatever is in row  $k$ .
  - To return `false` if all remaining equations have 0's in column  $k$ . The false is then returned to `solve()` to indicate that the system of equations has no solution.
- The first task is to select an equation, called the *pivot equation*, for the next phase of the process. On pass  $k$ , we look at all of the equations in rows  $k$  and greater. We select the equation with the largest coefficient in column  $k$  because this maximizes computational accuracy.

---

The `swapNonzeroRow()` function is called from `solve()` in Figure 18.33.

```

bool Gauss:: swapNonzeroRow( int k ) {
    int row = findPivot( k );
    // Coefficient k of pivot equation must be non-zero. -----
    if ( fabs( mat[row][k] ) < EPS ) return false;
    swap( mat[row], mat[k] );
    print( cout, "After swap." );                                // For debugging.
    return true;
}

// -----
// Find the equation with the largest coefficient in column k.
int Gauss:: findPivot( int k ) {
    double coefficient = fabs( mat[k][k] );           // Largest value so far.
    int big = k;                                         // Index of current coefficient.

    for (int row = k+1; row<nEq; ++row) {
        if (fabs( mat[row][k] ) > coefficient) {
            coefficient = fabs( mat[row][k] );      // A bigger coefficient.
            big = row;                            // Remember where it was found.
        }
    }
    return big;          // Line number of equation with biggest coefficient.
}

```

---

**Figure 18.34. Finding the next pivot row.**

- `swapNonzeroRow()` delegates to the `findPivot()` function the job of finding the coefficient in column  $k$  that is farthest from zero.
- All comparisons must be made using the absolute value of the coefficient. `fabs()` is the C-standard function for computing the absolute value of a floating point number (type `double` or `float`).
- `findPivot()` calculates the maximum value in column  $k$  and returns the subscript of the row it is in. That row is swapped with the current row and becomes the next pivot equation.
- If the largest coefficient in column  $k$  in the remaining equations is zero or very close to zero, dividing by it will cause floating point overflow. If this happens, the problem cannot be solved by this method, either because it contains inconsistent formulas, or because one of the equations is a linear combination of some of the others. So `swapNonzeroRow()` checks for this and returns an error code.
- If one of the equations is a linear combination of some of the others, there are not enough constraints to determine a unique solution. If the equations are inconsistent, there are too many constraints and they cannot be simultaneously satisfied.
- The constant `EPS` defines what we mean by “too close to zero”. The output below shows the beginning and end of the output from running the program on an unsolvable system of equations.

Linear Equations to Solve

---

1.000 * a +	2.000 * b +	2.000 * c +	1.000 * d +	10.000
3.000 * a +	-1.000 * b +	0.000 * c +	5.000 * d +	-4.000
2.000 * a +	2.000 * b +	3.000 * c +	0.000 * d +	15.000
2.000 * a +	2.000 * b +	3.000 * c +	0.000 * d +	14.000

---

Starting solve loop at pivRow 0

...

---

This function is called from Figure 18.33.

```
// Scale kth equation in matrix to have 1 in kth place
// Assumes places 1..k-1 are already zero.
// -----
void Equation:: scale( int col ) {
    double factor = coeff[col]; // scale factor
    for (int k=col; k <= coeff.size(); ++k)
        coeff[k] /= factor;
}
```

---

**Figure 18.35.** Equation::scale().

#### Equations after Elimination

```
1.000 * a + 0.000 * b + 0.000 * c + 2.600 * d + -1.600
0.000 * a + 1.000 * b + 0.000 * c + 2.800 * d + -0.800
0.000 * a + 0.000 * b + 1.000 * c + -3.600 * d + 6.600
0.000 * a + 0.000 * b + 0.000 * c + 0.000 * d + -1.000
```

---

Equations inconsistent or not independent.

**Second inner box of Figure 18.33 and Figure 18.35: Scale an equation.** Remember: Dividing every term of an equation by the same number does not change the solution of the equation.

- We divide all terms of the equation in row  $k$  by the value in column  $k$ . This leaves a value of 1 in `matrix[k][k]`.
- The loop starts at column  $k$  because all prior columns in row  $k$  are zero .

**Third inner box of Figure 18.33 and Figure 18.36: Wipe an equation.** After scaling row  $k$ , there is a 1 in `matrix[k][k]`.

- The next step is to use that 1 to wipe out all the coefficients below it in column  $k$ .
- Take  $c$ , the coefficient in column  $k$  of row  $m$ . Multiply the entire pivot equation by  $c$ , then subtract the result from the equation in row  $m$ . This row operation does not change the solution to the system.
- Repeat this process for every equation in the matrix except the pivot equation. Now there are 0's in column  $k$  in every row except the pivot row.

## 18.5 What You Should Remember

### 18.5.1 Major Concepts

- The nature of control structures follows the form of the data being processed. Using 1D arrays requires a single loop control structure. Using 2D arrays requires nesting one loop within another to process all data elements. This effect continues as the data complexity increases.

---

This function is called from Figure 18.33.

```
// Clear coefficient k1 of equation k2 by subtracting mat[k2][k1] * equation k1
// from equation k2. Assumes that mat[k1][k1] == 1.
// -----
void Equation:: wipe( Equation& piv, int pCol ) {
    double factor = coeff[pCol];
    for(int col=pCol; col<=coeff.size(); ++col) {
        coeff[col] -= factor * piv[col];
    }
}
```

---

**Figure 18.36.** Equation::wipe().

- A `typedef` name stands for the entire type description, including all the asterisks and array dimensions. In the basic syntax, this information is scattered throughout a variable declaration, with variable names written in the middle. This makes C declarations hard to read and easy to misunderstand. When you use a `typedef` name, all the type information goes at the beginning of the line and all the variable names come at the end.
- A table of data can be represented conceptually in two different manners. An array of arrays implies a unity of the elements in each of the rows in the table, whereas a matrix implies that each element in the table is independent of the others.
- Whether declared as an array of arrays or a matrix, the data elements are still stored sequentially in memory in row-major order. This layout can be exploited when filling or removing data from the structure, as in reading and writing data from files using `read()` and `write()`.
- The C language allows for arrays of many dimensions, although in practice using more than three or four dimensions is unusual.
- Example applications of 2D arrays include matrix arithmetic and image processing. Matrix arithmetic uses both 1D vectors and 2D matrices in various calculations. Image processing programs use 2D arrays of various sizes and typically store the data in binary files.

***Two kinds of matrix.*** C++ supports two distinctly different dynamic implementations of a matrix. In one implementation, all of the memory is contiguous and allocated by a single call on `new`. In the other, the matrix is represented as an array of pointers or a vector of vectors, each pointing at an array of data elements. Here, `new` is called several times, once for the dynamic array of pointers and once for each of the dynamic arrays of elements.

***The dynamic 2D array.*** A dynamic 2D array is useful for applications such as image processing, in which data sets of varying sizes will be processed. The size is generally known at the beginning of run time and a properly sized block of storage can be allocated then. Unfortunately, the programmer has to write a function to do the address calculations for converting the conceptual 2D subscripts to a physical 1D subscript before accessing a particular element. This calculation is examined in the exercises.

***The dynamic matrix data structure.*** A dynamic matrix can be an array of pointers to arrays. Like static two-dimensional arrays, elements are accessed by using two subscripts. The first subscript selects a row from the array of pointers, the second selects a column from that row. This data structure is appropriate if rows of the matrix must be interchanged.

***A matrix made of vectors.*** A dynamic matrix can also be a vector of vectors because the `vector` class supports the subscript operator.

***Gaussian elimination.*** Gaussian elimination is a well-known method for solving systems of simultaneous linear equations in several unknowns. The algorithm is easily implemented using a dynamic matrix data structure and a set of functions that perform row operations on the equations.

### 18.5.2 Programming Style

- Avoid deeply nested control structures. If the nesting level is greater than three, the logic can be very difficult to follow. It is better to break up the code by defining a new function that performs the actions of the inner loops on the particular data items selected by the outer loops.
- The processing of data in a 2D array typically is done using two `for` loops. Other loop combinations can be used, but the double `for` loop has become almost standard.
- The programmer must take care when using nested loops to make sure that the outer and inner loops each process the appropriate dimension of a 2D array. It helps a great deal to use meaningful identifiers for the subscripts, such as `row` and `col`, rather than the simpler `j` and `k`.
- In a set of nested loops, the number of times the innermost loop body is executed is the product of the number of times each loop is repeated. It is important to make the innermost statements efficient, since they will occur many times.

- Use `#define` appropriately. As with 1D arrays, defining the array dimensions as constants at the top of a class makes later modification simple.
- Using `typedef` to name the parts of nested array types makes the code correspond to your concepts. This makes it easier to write, compile, and debug a program, because it allows you to declare parameters with the correct types and enables the compiler to help you write the appropriate number of subscripts on array references and arguments.
- Any legal C identifier may be used as the type name in a `typedef`. Some caution should be used, however, to avoid names that sound like variables or keywords. Types are classes of objects, so the type name should be appropriate for the general properties of the class.
- Usually a choice must be made as to whether to use an array of arrays or a matrix. If each of the data elements is independent of the others, then the matrix is the appropriate structure. If the data in a single row have a meaning as a group, then it is correct to use the array of arrays. When data in a column also have a meaning, either structure can be used.
- Continuing the preceding reasoning, always pass the proper parameter. When using a function to process a multidimensional array, pass only the part of the array that is needed, if possible. When a function works on all elements of the array or on an entire column, the parameter should be the entire array. However, if a function processes only one row, simply pass the single row, not the whole matrix. And if a function processes a single array element, pass just that element.

### 18.5.3 Sticky Points and Common Errors

- When using nested loops, it is fairly common to write a statement at the wrong nesting level. The action then will happen either too often or not often enough. Use curly brackets where necessary to make sure statements are in the correct loop.
- As always, all subscripts, even in a multidimensional array, start at 0, not 1.
- The programmer must be careful always to use subscripts within the bounds of the array. When using a 2D array, it is possible for a column subscript to be too large and still have the referenced element be within the memory area of the array. This is a serious error and usually harder to find than simply referencing outside the array's memory.
- Always beware of using subscripts in the wrong order. This may cause you to access outside the array's memory, or it might not. Using meaningful subscript names reduces the frequency of these errors.
- It also is important to use the proper number of subscripts. Because legally you can reference a single row in a matrix, most such references are not deemed incorrect by the compiler and may generate only a warning. However, rather than get the data item you desire, you will be using a memory address in your calculations.
- Use correct C++ syntax: `a[row][col]` rather than the syntax common in other languages: `a[row, col]`.

### 18.5.4 New and Revisited Vocabulary

These are the most important terms, concepts, keywords, and C library functions discussed in this chapter:

array <code>typedef</code>	vector of vectors	digital image
dimension	double subscripts	pixel
vector	column	smoothing
matrix	row	processing window
2D and 3D initializers	row-major order	2D transformation
multidimensional array	binary file mode	plane
array of arrays	<code>read()</code>	translate
array of strings	<code>write()</code>	rotate
dynamic 2D array	binary data	system of equations
dynamic array of arrays	nested loop	Gaussian elimination

## 18.6 Exercises

### 18.6.1 Self-Test Exercises

1. Assume that `mat` is an  $N$  by  $N$  matrix of `floats`. Declare two pointers, `begin` and `end`, and initialize them to the first and last slots of `mat`. Declare a third pointer, `off`, and initialize it as an off-board pointer.
2. Given the declarations and code below, show the output of the loops.

```
#define Z 3
char square[Z][Z];
char* p;
char* start = &square[0][0];
char* end = &square[Z-1][Z-1];

for (p = end; p >= start; p -= 2) *p = '1';
for (p = start; p < end; ++p) {
    ++p;
    *p = '0';
}

for (p = start; p <= end; ++p) cout <<*p <<" ";
cout << endl;
```

3. Assume your program contains this code:

```
short int box[5][4];
int j, k;

for (k = 1; k <= 3; k++)
    for (j = 1; j <= 3; j++)
        box[j][k] = 2*j-k;
```

- (a) Draw a flowchart of this code.  
(b) Make a diagram of the array named `box` showing the array slots and subscripts. In the diagram, write the values that would be stored by the `for` loops. If no value is stored in a slot, write a question mark there.  
(c) What are the subscripts of the slot in `box` in which the value 5 is stored?  
(d) What is `sizeof box`?  
(e) Which array slot will have the lower memory address, `box[1][2]` or `box[2][1]`? Why?  
(f) What happens if you execute this line: `box[5][4] = 10;`?
4. Diagram the array created by the following declarations. In the drawing, identify the slot numbers and show the values stored in those slots by the loops that follow. Also, write a new declaration statement that has an initializer to set the contents of the array to that produced by these loops.

```
#define Y 4
#define Z 3
int a, b;
float lumber[Z][Y];

for (a = 0; a < Z; ++a)
    for (b = 0; b < Y; b++) {
        if (b > a)         lumber[a][b] = 0;
        else if (b == a)   lumber[a][b] = 2.5;
        else               lumber[a][b] = (a + b) / 2.0;
    }
```

5. Write a function, `findMin()`, that has one parameter, `data`, that is a 10-by-10 matrix of `floats`. The function should scan the elements of the matrix to determine the element with the minimum value, keeping track of its position in the matrix. Return the value itself through the normal return mechanism and its row and column subscripts through pointer parameters.
6. Trace the execution of the following code using a table like this one:

m	
k	
Output	

Show the initial values of the loop variables. For each trip through a loop, show how these values change and what is displayed on the screen. In the table, draw a vertical line between items that correspond to each trip through the inner loop.

```
int k, m;
int data[4][3] = { {1,2,3}, {2,4,6}, {3,6,9}, {4,8,12} };

for (k = 0; k < 4; ++k) {
    for (m = k; m < 3; ++m)
        if (k != 1) cout << " " << data[m][k] ;
        else         cout << " " << data[k][m] ;
    cout << '\n' ;
}
```

7. Trace the execution of the following loop, showing the output produced. Use the array `data` and initial values declared in the previous exercise. Be careful to show the output formatting accurately.

```
for (k = 0; k < 3; ++k) {
    for (m = 0; m < 3; ++m) {
        if ((k+m) % 2 == 0) cout << data[m][k];
        else                 cout << "   ";
    }
    cout << '\n';
}
```

8. Happy Hacker wrote and printed the following code fragment in the middle of the night, when some of the keys on his keyboard did not work. The next day, he looked at it and saw that it was not quite right: All of the curly brackets were missing, there was no indentation, there were too many semicolons, and the loops were a little messed up. The code is supposed to print a neat matrix with three columns and seven lines. In addition, the column and row numbers are supposed to be printed along the top and left sides of the matrix. Please fix the code for Happy.

```
#define Z 3
#define W 7
int k, m;

cout << "   ";
for (k = 0; k < Z; ++k);
cout << setw(2) << k;
cout << "\n ----- \n" ;
for (m = 0; m < Y; ++m;
cout << setw(4) << m;
for (k = 0; k < W; ++k;
cout << setw(2) << mat[m][k];
cout << '\n' ;
```

### 18.6.2 Using Pencil and Paper

1. The pair of loops that follow initialize the contents of a 2D array, `data`. Draw a picture of this array, labeling the subscripts and showing the contents of the slots after the loops are done.

```
int j, k;
int data[4][3];

for (k = 0; k < 4; k++)
    for (j = 0; j < 3; j++)
        data[k][j] = j * k + 1;
```

2. Trace the following loop and show the output exactly as it would appear on your screen:

```
#define Z 3
int a, b;
char square[Z][Z+1] = { "cat", "ode", "dog" };

for (a = 0; a < Z; ++a) {
    cout << a;
    for (b = 0; b < Z; b++) {
        if (a == 0) cout << setw(2) << square[a][b];
        else cout << setw(2) << square[b][a-1];
    }
    cout << '\n';
}
```

3. Write a function, `countZeroRows()`, with one matrix parameter. It should count and return the number of rows in the matrix that have all zero values. Assume the matrix is a square array of size  $N$ -by- $N$  integers, where  $N \geq 1$  and is defined as a global constant.

4. Given the declarations and code that follow, show the output of the loops:

```
int k, m;
int data[4][3] = { {1,2,3}, {2,4,6}, {3,6,9}, {4,8,12} };

for (k = 0; k < 4; ++k) {
    for (m = 0; m < 3; ++m) {
        if ((k*m)%2==0) cout << setw(2) << data[k][m];
        else cout << "    ";
    }
    cout << '\n' ;
}
```

5. Given the declarations and code that follow, trace the code and show the output of the loops:

```
#define Z 3
char square[Z][Z+1] = { "---", "---", "---" };
int r, c;

for (r = 2; r >= 0; --r) square[r][r] = '0';
for (r = 0; r < 3; ++r) square[r][(r + 1)%Z] = '1';
for (r = 0; r < 3; ++r) {
    for (c = 0; c < 3; ++c) cout << square[r][c];
    cout << '\n';
}
```

6. For each item that follows, write two C++ statements that perform the task described. Write the first using subscripts, the second using pointers and dereference. Explain which is clearer and easier to use and why. Use these declarations:

```

const int rows = 5, cols = 3;
int k, m;
float A[cols];
float M[rows][cols];
float* N = new float[(rows * cols)];
float* p, * end;

```

- (a) Make `p` refer to the first slot of `A`.
- (b) Write a `for` loop to set all elements of `A` to one.
- (c) Double the number in `M[4][2]` and store it back in `M[4][2]`.
- (d) Set `end` as an off-board pointer for `M`.
- (e) Make `p` refer to the first `float` in `N`.

### 18.6.3 Using the Computer

#### 1. Changing a figure.

Extend the crown program (point transformation) to permit the user to add a point to the figure between any two existing points or delete any point in the figure after it is originally constructed. Provide a menu so that the user can do several point transformations. When “quit” is selected, write out the new data set.

#### 2. Sales Bonuses.

A company with three stores, A, B, and C, has collected the total dollar amount of sales for each store for each month of the year. These numbers are stored in a file, `sales.in`, that has the information for January, then February, and so on. Within each month, the sales amount for store A is first, then store B, then store C.

Write a function, `readFile()`, that will read in the data from the sales file and store the amounts in a matrix where each row represents a store and each column represents a month.

The company pays each store a \$1,000 bonus if its sales exceed \$30,000 in any particular month. Next, write a function, `bonus()`, that has one parameter, a matrix of `float` values. It returns an integer to the caller. In this function, search through the `sales` matrix, starting at the first month for the first store. Test each value in the matrix to see if the sales amount qualifies for a bonus. If it does, print the store name and month corresponding to the row and column, as well as the sales amount. Count and return the number of bonuses given. If no amount qualifies for a bonus, return 0.

Finally, write a main program that first reads in all the data, then prints the sales amounts and total sales for each store in a neat table, with all of store A’s sales first, followed by those of store B, and finally those of store C. Last, call the `bonus()` function to determine the bonuses for the year. Print the total sales amount, the number of bonuses awarded, and the total amount of bonus money that the company will give out.

#### 3. Array averages.

An instructor teaches course of all sizes, and needs to compute grade averages. He typically gives between 5 and 10 quizzes each term.

- (a) Define two classes: Course and Student. A Course will be represented by a float (the overall quiz average), the number of quizzes for the term, and a vector of Students.

A Student has a `name`, a vector of `scores`, and `anaverage`, where each score is an integer between 0 and 200. The Student constructor should accept one parameter, the number of quizzes. It must read the name and quiz scores for one student, compute the average, and store use the data to initialize a new Student.

The Course constructor should open an input file and, call the Student constructor to read students, and store all the Students in a vector.

- (b) Write a main program that will compute and print the grade averages. Call the Course constructor and the `computeAverage()` function described below. Finally, print each student’s name, quiz grades, and quiz average in a neat table, followed by the overall average of all students on all quizzes.

- (c) The function `Student::computeAverage()` should compute the grade average for one student and store it in the current `Student`.
- (d) The function `Course::computeAverage()` should compute the grade average for the entire class and store it in the `Course` object.
4. Matrix transpose.

- (a) Define a class `Matrix`. Its data members should be the number of rows,  $R$ , number of columns,  $C$ , and two-dimensional  $R \times C$  array of `doubles`.
- (b) Define `SIZE` as a constant. The `Matrix` constructor should take  $R$ ,  $C$ , and a file name as parameters and read in an  $R \times C$  matrix of numbers from the specified file.
- (c) In the `Matrix` class, write a function, `print()`, to print out a matrix in neat rows and columns. Assume that  $C$  is small enough that an entire row will fit on one printed line.
- (d) In the `Matrix` class, write a function, `transpose()`, to perform a matrix transpose operation, as follows:
- If  $R \neq C$ , abort and print an error comment. You can only transpose square matrices.
  - To transpose a matrix, swap all pairs of diagonally opposite elements; that is, swap the value in  $M[I][J]$  with the value in  $M[J][I]$  for all pairs of  $I < SIZE$  and  $J < SIZE$ . The transposed matrix will replace the original matrix in the same memory locations.
  - Try to accomplish the operation with a minimal number of swaps. Warning: This cannot be done by the usual double loop that executes the body  $I \times J$  times.

When the operation is over, the original matrix should have been overwritten by its transpose. Elements with the same row and column subscripts will not change. As an example, the 3-by-3 matrix on the left is transposed into the matrix on the right:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

- (e) Write a main program to instantiate your `Matrix` class and test your functions. Print the matrix before and after you transpose it.
5. Matrix multiplication.

- (a) Define a class `Matrix` as in parts (a) through (c) of the previous problem. Add two more methods to this class:
- A constructor with two integer parameters  $M$  and  $N$  (no file name). Allocate but do not initialize space for a 2D  $M \times N$  array of doubles (the answer).
  - A function, `multiply()`, to perform a matrix multiplication operation, as follows:
    - There will be two matrix parameters, `A` and `B` of the sizes just mentioned. The implied parameter will be `C`, the result matrix.
    - The result elements of `C` are defined as:

$$C_{ij} = \sum_{k=0}^{P-1} A_{ik}B_{kj}$$

where  $i$  is in the range  $0 \dots M - 1$  and  $j$  is in the range  $0 \dots N - 1$ . This essentially is the dot product operation shown earlier in the chapter. An example calculation would be

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \times \quad \begin{bmatrix} 4 & 7 \\ 5 & 8 \\ 6 & 9 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 32 & 50 \\ 77 & 122 \end{bmatrix}$$

- (b) This problem will work with three matrices of sizes  $M \times P$ ,  $P \times N$ , and  $M \times N$ . Write a main program that will instantiate matrices `A` and `B` from user-specified files using `#defined` constants  $M$ ,  $P$ , and  $N$ . Create matrix `C` for the results. Perform matrix multiplication on them, write the results to a file. Display the input matrices on the screen, then display the result matrix.

## 6. Bingo.

In the game Bingo, a playing card has five columns of numbers with the headings *B*, *I*, *N*, *G*, and *O*. There are five numbers in each column, arranged randomly, except that all numbers in the *B* column are between 1 and 15, all numbers in the *I* column are between 16 and 30, *N* has 31–45, *G* has 46–60, and *O* has 61–75. The word *FREE* is in the middle slot in the *N* column instead of a number. Your job is to create and print a Bingo card containing random entries in the proper ranges. Define a type, `card`, to be a 5-by-5 array of integers. Generate 25 random numbers and use them to initialize a card, `C`, as follows:

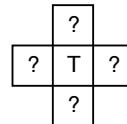
- (a) Declare an array of length 15 named `ar`. Repeat the next step five times, with  $x = 0, \dots, 4$ .
- (b) Store the number  $(1 + 15 \times x)$  in `ar[0]`, then fill the remaining 14 slots with the next 14 integers. Repeat the next step five times, once for each of the five rows on the Bingo card.
- (c) Randomly select a number, `r`, between 1 and 15. If the number in `ar[r]` is not 0, copy it into column  $x$  of the current row of the Bingo card and store 0 in `ar[r]` to indicate that the number already has been used. If `ar[r]` already is 0, generate another `r` and keep trying until you find a nonzero item. This trial-and-error method works adequately because the number of nonzero numbers left in the array always is much larger than the number of zero items. You would have to be very unlucky to hit several used items one after another.
- (d) When all 25 random numbers have been filled in, store a code that means *FREE* in the slot in the middle of the card. Print the resulting Bingo card as follows, with spaces between the numbers on each row. You may use lines of minus signs for the horizontal bars and omit the vertical bars

<b>B</b>	<b>I</b>	<b>N</b>	<b>G</b>	<b>O</b>
3	17	32	49	68
12	23	38	47	61
11	18	FREE	60	70
2	29	36	50	72
9	22	41	57	64

## 7. Crowded items.

Define  $R$  and  $C$  as global constants of reasonable size. Define a class, `matrix`, with  $R$  rows and  $C$  columns of type `unsigned char` elements.

- In the Matrix class, write a function, `findCrowdedElements()`, with `amatrix` parameter for output. The implied parameter will be the input to this process.  
The values in the input matrix will be either 1 or 0, representing `true` and `false`. The function will set each value of the output matrix to 1 or 0 according to whether the corresponding element in the input matrix is “crowded.”
- Crowded is defined thus: First, an element’s value must be `true` for it to be crowded. Second, there are potentially four meaningful neighbors of the test element (up, down, left, right):



- If three or more of the current element’s four neighbors have a value of `true`, then the current element is crowded.
- If the current element is on a border, but not in a corner, then only two neighbors have to be `true` to be crowded.
- Corner elements need only one of their two neighbors to be `true` to be crowded.

The diagram illustrates a transformation of a 5x5 matrix. On the left, the input matrix has rows labeled 0 through 4 and columns labeled 0 through 4. The values are:

0	T	T	F	F	F
1	T	T	T	T	F
2	T	F	T	T	T
3	F	T	F	T	T
4	F	T	F	T	F

On the right, the output matrix has rows labeled 0 through 4 and columns labeled 0 through 4. The values are:

0	T	T	F	F	F
1	T	T	T	F	F
2	F	F	F	T	T
3	F	F	F	T	T
4	F	F	F	F	F

- Write a main program. Input a matrix of values from a file, compute the “crowded” matrix, and write it out to another file. Assume the input file contains data for a matrix of the proper size. In addition to saving the result in a file, display output similar to that above, using ‘F’ and ‘T’.

8. Bar graph.

A professor likes to get a visual summary of class averages before assigning grades. Her class averages are stored in a file, `avg.in`. You are to read the data in this file and from it make a bar graph, as follows:

- Your graph should have 21 lines of information, which are printed with double spacing.
- Down the left margin will be a list of 5-point score ranges: 0...4, 5...9, up to 100...104. Following this, on each line, should be a single digit (defined later) for each student whose average falls within that range. There may be scores over 100 because this professor sometimes curves the grades.
- Any negative scores should be treated as if they were 0. Any scores over 104 should be handled as if they were 104.

Represent this graph as a matrix of characters with 21 rows and 35 columns initialized to 0 (null characters). You may assume that the number of scores put into any single row will not exceed 35. Also, have an array of integer counters corresponding to the rows. As each average is read, determine which row,  $r$ , it belongs in by using integer division. Also, isolate the last digit,  $d$ , of each score; that is, for 98,  $d = 8$  and for 103,  $d = 3$ . To record the score, convert digit  $d$  to the ASCII code for the digit by adding it to ‘0’. Store the resulting character in the next available column of row  $r$  and update the counter for row  $r$ . When the end of the file is reached, print the matrix on the screen using a two-column field width for each score, and a blank line between each row.

9. Local maxima in a 2D array. Assume that you have a data file, `max.in`, containing 1-byte integers in binary format. The file has 50 numbers, representing a matrix of values that has five rows containing 10 numbers each. The numbers are stored in a row-major order. Write a program that will do the following:

- Read the data values into a 2D array on which further processing can be performed.
- Call the `localMax()` function described next and print the values of both the input and output matrices.
- Write a function, `localMax()`, that takes two 2D arrays of integers as parameters. The first parameter is the input matrix; the second is an output matrix. Set the value at a particular location in the output matrix to 1 if the corresponding input value is a “local maximum”; that is, the value is larger than the four neighboring values to its left, right, above, and below. Set the output value to 0 if the corresponding position cannot be labeled as a local maximum. Matrix positions on a corner or an edge will have only two or three neighboring points, which must have lesser values for the given location to be a maximum. As an example, the input matrix on the left would generate the matrix on the right:

0	6	9	7	5	6		0	1	0	0	1
1	2	6	3	3	4		0	0	0	0	0
2	2	2	4	1	7		0	0	0	0	1
3	6	5	9	5	6		1	0	1	0	0
4	2	6	4	6	8		0	1	0	0	1
	0	1	2	3	4		0	1	2	3	4



## 10. Local maxima and minima.

In solving this problem, refer to the smoothing program in Figure 18.17 and generate a new program to do the following:

- (a) Open and read a user-specified ASCII file that contains a  $10 \times 10$  matrix of integer values in the range 0–9 and store them in a 2D array.
- (b) Then generate a corresponding  $10 \times 10$  array of character values, each of which is determined by the following:
  - i. Generate a '+' if the value in the original array is a local maximum, where *local maximum* is defined as the highest value in a  $3 \times 3$  area centered on that position in the matrix. For values on the border, examine only the portion of this area containing actual values.
  - ii. Generate a '-' if the value in the original array is a local minimum, where *local minimum* is defined as the lowest value in the same  $3 \times 3$ .
  - iii. Generate a '\*' if the value in the original array is a saddle point, where *saddle point* is defined as a value higher than the neighboring values in the same column but lower than the neighboring values in the same row. Saddle points cannot occur on a border.
  - iv. Finally, if the value is not classified as any of the preceding three, simply convert the integer value of 0–9 into a character '0'–'9' by adding the number to the character value '0'.
  - v. As an example, using a  $5 \times 5$  array,

$\begin{bmatrix} 0 & 2 & 3 & 2 & 4 \\ 6 & 5 & 6 & 0 & 3 \\ 7 & 4 & 1 & 2 & 3 \\ 5 & 8 & 7 & 4 & 8 \\ 1 & 2 & 9 & 3 & 2 \end{bmatrix}$	$\rightarrow$	$\begin{bmatrix} - & 2 & 3 & 2 & + \\ 6 & * & + & - & 3 \\ 7 & 4 & 1 & 2 & 3 \\ 5 & 8 & 7 & * & + \\ - & 2 & + & 3 & - \end{bmatrix}$
---	---------------	---

- (c) Display both the numeric matrix and the character matrix on the screen.

## 11. Computing the wind chill.

The wind-chill index is a measure of the increase in heat loss by convection from a body at a specific temperature. Consider the wind-chill table below.

		Actual Air Temperature (F)									
		-10	-5	0	5	10	15	20	25	30	35
Wind Speed (mph)	5	5	5	5	5	4	4	4	3	3	2
	10	24	22	22	20	19	18	17	16	14	13
	15	35	35	33	30	28	26	26	24	19	19
	20	42	41	40	37	34	32	29	29	27	23
	25	48	47	45	42	39	37	35	32	30	28
	30	53	51	49	46	43	41	38	36	32	30
	35	57	55	52	48	45	42	40	38	34	32

In `main()`, instantiate `Chill` (below) then enter a query loop that will call a function, `getChill()`, to perform the main process as many times as the user wishes. Print instructions and prompt the user to enter real values for the actual temperature,  $t$ , and the wind speed,  $s$ .

Define a class `Chill` to contain this table and the related functions. The constructor must read the wind-chill table from a text file, `chill.in`, and store it in a two-dimensional array (the column and row headings are not in the data file).

The `getChill()` function should have the prototype `void chill( double t, double s )`. Within the `chill()` function do the following:

- (a) Validate  $s$  and  $t$ . Print an error comment and return if  $s$  is not in the range  $2.5 \leq s < 37.5$  mph or  $t$  is not in the range  $-12.5^\circ \leq t < 37.5^\circ$  Fahrenheit. Use these values to look up the effective temperature decrease in the wind chill table as follows.
- (b) Calculate the row subscript. For any input wind speed,  $s$ , we want to use the closest speed that is in the table. If  $s$  is a multiple of 5 mph, the table entry is exactly right. If not, we must calculate the closest speed that is in the table. For example, if the speed is between 7.5 and 12.5 mph, we want to use the row for 10 mph, which is row 1. To calculate the correct row number, subtract 2.5 from  $s$ , giving a result less than 35.0. Divide this by 5.0 and store the result in an integer variable. Since fractional parts are discarded when you store a `float` in an integer, the integer will be between 0 and 6 and usable as a valid row number.
- (c) Calculate the column subscript. As for the wind speed, use the temperature in the table closest to the input temperature,  $t$ . For example, if the temperature is between  $-2.5^\circ$  and  $+2.5^\circ$ , use the values for  $0^\circ$ , which are in column 2. To calculate the correct column number, add 12.5 to  $t$ , giving a positive number less than 50.0. Divide this by 5.0 and store the result in an integer variable. This will give an integer between 0 and 9 that can be used as a valid column number.
- (d) Use the computed row and column subscripts to access the wind chill table and read the decrease in temperature. To compute the effective temperature, subtract this decreased amount from the actual air temperature. Echo the inputs and print the effective temperature.

## 12. Matrix averages.

Write a program that will input a matrix of numbers and output the row and column averages. More specifically,

- (a) Read a file containing two integers on the first line: a number of rows and a number of columns. Then allocate memory for a matrix of `floats` with those dimensions. Read the rest of the data from the file, row by row, into the matrix.
- (b) Allocate an array of `floats` whose length equals the number of rows and store the average of each row in the corresponding array slot.
- (c) Allocate another array with one slot per column and store the average of each column in the corresponding array slot.
- (d) Print the matrix in spreadsheet form, with row numbers on the left, row averages on the right, column numbers on top, and column averages at the bottom.

Use arrays, not vectors.

## **Part VI**

# **Developing Sophistication**



# Chapter 19

# Recursion

This chapter presents **recursion**, a fundamental technique in which a function solves a problem by dividing it into two or more sections, calling itself to solve each portion of the original problem, continuing the self-calls until a subproblem's solution is trivial, and then building a total solution out of the partial solutions. We examine how to apply recursion to two familiar problems: searching and sorting.

Prior to this, as an aid to understanding how recursion works, we briefly survey C/C++ storage classes, giving a detailed description of how the `auto` storage class is used to implement recursive parameter passing.

## 19.1 Storage Classes

Every variable has a storage-class attribute in addition to its data type and perhaps a type qualifier like `const` or `volatile` as well. The **storage class** determines when and where a variable is allocated in memory, when it is initialized, how long it persists, and which functions may use it. There are three useful storage classes in C++: `automatic`, `extern`, and `static`. `Automatic` is the default storage class for all local variables and parameters. `Static` is used to create a variable with a lifetime longer than the block that declares it. `Extern` is the default for global variables. It makes a global variable in one module visible to other program modules. `Extern` is rarely needed if good OO design is used.

### 19.1.1 Automatic Storage

The automatic storage class is used in C/C++ for function parameters. Local variables within functions also are automatic by default, but can be declared `static`, as discussed in the next section. Whenever you call a function, memory for every automatic variable is allocated in an area called the **run-time stack**, with other function call information. This memory is deallocated when the function returns. The stack memory is managed automatically and efficiently by the C/C++ system.

Automatic is the default storage class for local variables because it makes the most efficient use of computer memory. Good programmers use automatic variables (rather than `static` local variables or global variables) wherever possible, because their lifetime is brief and their scope is restricted to one function. They provide an effective way to isolate the actions of each program unit from the others. Unintentional interference between functions is minimized, making it much easier to debug a large program. On the other hand, the constant allocation and deallocation of storage adds slightly to the execution time. Because it is done quite efficiently, the time wasted is minimal and the advantages of the added modularity are great. The characteristics of automatic memory are discussed in greater detail in an upcoming section on the run-time stack, but briefly automatic memory is

- **Allocated** on the run-time stack.
- **Initialized** during the function call.
- **Visible** only to the statements in the body of the declaring function.
- **Deallocated** during the function-return process. Local variables are freed by the called function and parameter storage is freed by the caller, both under the control of the C/C++ run-time system.

### 19.1.2 Static Storage

The name *static* is derived from the word *stay*; it is used because **static** variables stay around from the beginning to the end of execution. In older computer languages, all variables were **static**. In C/C++, local variables are automatic by default but can be declared **static**. There are two purposes for **static** local variables: to remember the state of the subprogram from the end of one execution to the beginning of the next, and to increase time and space efficiency when a function uses a table of constants. The characteristics of **static** local memory are

- It is allocated in a portion of computer memory separate from the run-time stack, which we call the **static allocation area**. In many implementations, this immediately follows the memory block used for the program's code.
- It is initialized when the program is loaded into memory. Initializers must be constants or constant expressions; they cannot refer to non-**static** variables, even ones with a **const** qualifier.
- It is deallocated when the program exits and returns to the system.
- It is visible to the statements in the body of the function in which the **static** variable is declared.

**Global variables** default to **extern** but also may be declared **static**.<sup>1</sup> A **static** global variable is like a **static** local variable in almost every way. It is declared at the top of the source-code file and visible to all functions in that same file.<sup>2</sup>

**The state of a function.** In some complex situations, a function must remember a piece of data from one call to the next. For example, a function that assigns consecutive identification numbers to a series of manufactured objects would need to know the last number assigned before it could assign the next. We say that this last number is part of the **state** of the system. Since automatic variables come and go as function calls are begun and completed, they cannot be used to store state information. Global variables can be used, but doing so is foolish because these variables can be seen and changed by other parts of the program. A **static** local variable is an ideal way to save state information because it is both private and persistent.

**Tables of constants.** Normally, the time consumed by the allocating and initializing of automatic variables is minimal. It is not considered a waste because this allocation method guarantees that every function starts its computations with a clean slate. However, reinitializing a **table of constants** every time you enter a function that uses them can consume appreciable execution time. It also requires extra space, because the program code must contain a copy of the entire initializer for the table, which it uses to reinitialize the local automatic table every time the function is called. If the table truly is constant, there is certainly no need to keep recopying it.

We could solve this problem by declaring the table as a global constant. However, this is not really a good solution if the table is used by only one function. A general principle is that things used together should be written together in a program. If possible, a table should be defined by the function that uses it, not at the top of the program file, which may be several pages away. A better way to eliminate the wasted time and space is to declare the table inside the function with a **static** storage class. It will be visible only within the defining function and will not be re-created every time the function is called. Space will be allocated once (before program execution begins) and initialized at that time.

### 19.1.3 External Storage (Advanced Topic)

A variable or function of the **extern** storage class is made known to the linker explicitly. This becomes useful when an application is so large that it is broken up into modules, often written by different people. Each module is a set of types, functions, and data objects in a separate source-code file. These files are compiled separately and joined into a single working unit by the linker. In such a program, many objects and functions are defined and used only within one subsystem. Others are defined in one module and used by others. They must be made known to the linker, so that it can make the necessary connections. This is known as **external linkage**.

---

<sup>1</sup>This makes a difference only in a multymodule application where the modules are compiled separately and linked before execution.

<sup>2</sup>The **static** variable is visible to functions from any included files if the **#include** command follows the static declaration.

Functions, as well as global constants and variables, are labeled `extern` by default. These symbols are made known to the linker unless they are explicitly declared to be `static`. A `static` function or object is visible only within the code module that defines it.

**Rules for `extern` declarations.** Developing large, multi-module programs is beyond the scope of this text. The following details are provided for general knowledge and as a basis for the small multi-module system developed in Chapter 20.3.

An `extern` symbol must be declared in both the code file that supplies it and the code files that use it. One declaration defines the object and one or more declarations in other modules may import it. These two kinds of declarations are subtly different in several ways:

**Creation.** A data object declaration with an initializer is the defining occurrence, whether or not the keyword `extern` is present. A function prototype is a defining occurrence but only if the function definition appears in the same code module.

**Allocation and initialization.** The defining occurrence is allocated and initialized when its code module is loaded into memory. Initializers must be constants or constant expressions.

**Deallocation.** External symbols are global in nature, therefore they are freed when the program exits and returns to the system.

**Visibility.** The defining occurrence is visible in its own code module and any other code module that contains a matching declaration with the keyword `extern`. The importing occurrence must not include an initializer (if it is a data object) or a function definition (if it is a prototype).

## 19.2 The Run-Time Stack (Advanced Topic)

During a function call, the caller passes arguments that will be stored in the called-function's parameter variables, and these are used by the called-function in its computations. Static storage cannot be used for **parameters** because recursion requires storage for each parameter in the stack-area created for each call on the function. When a function calls itself, two or more sets of its argument values must coexist. At compile time, the translator cannot know how many times a recursive function will call itself, so it cannot know how much storage will be needed for all instances of its parameters. For this reason, C/C++ allocates space for parameters and local variables in a storage area that can grow dynamically as needed.

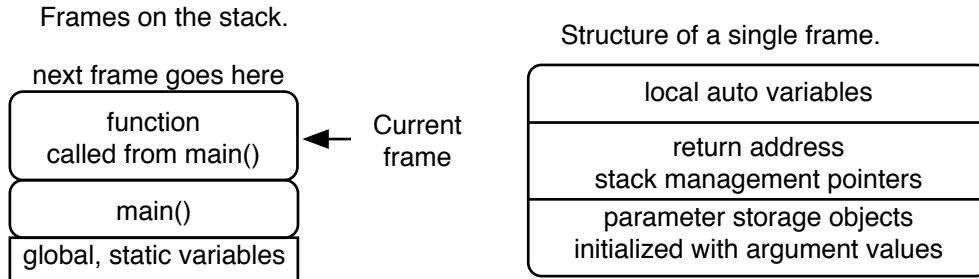
In C/C++, as in most modern languages, a data structure called the *execution stack*, *run-time stack*, or simply, *stack* is used to store one *stack frame* for each function that has been called, but has not yet returned, since execution began. In this section, we look closely at the mechanism by which a function actually receives and stores its parameters.

### 19.2.1 Stack Frames

A function's **stack frame**, or **activation record**, holds its parameters, return address, local variables, and a pointer to the prior stack frame. In it, there is one variable for each parameter listed in the function header. Before control is transferred from the caller to the function, the argument values written as part of the function call are computed and copied into the parameter variables in the new stack frame, as follows:

1. Memory is allocated in the stack frame for the parameters.
2. The argument expressions in the call are evaluated and the resulting values are written in order on the stack. If the type of an argument does not match the type of the corresponding parameter in the function's prototype, the argument value is converted to the required type.
3. Then control is transferred to the subprogram with a jump-to-subroutine machine instruction. One of the actions of this instruction is to store the return address in the stack.
4. When control enters the subprogram, it associates the values in the stack frame, in order, with the parameter names in the function declaration. More stack storage is then allocated and initialized for the local automatic variables (storage for `static` variables was allocated at load time).

The diagram on the left shows the arrangement of frames on the stack; the frame for `main()` is allocated first; a frame for each other function is added when the function is called and removed when it returns. The diagram on the right shows a more detailed view of the components of one stack frame.



**Figure 19.1. The run-time stack.**

5. Finally, control is transferred to the first line of code in the subprogram. Execution of the subprogram continues until a `return` statement is reached. If there is no `return` statement, execution continues until the last line of code is finished.

When execution of the function ends, the result is returned and storage is freed as follows:

1. The result of the `return` expression is converted (if necessary) to the return type declared by the function prototype. The return value then is stored where the caller expects to find it.
2. The local automatic variable storage area is freed. (Local `static` variables are not freed until the program terminates.)
3. Control returns to the caller at the `return address` that was stored in the stack frame during the function call.
4. The caller picks up the return value from its prearranged location and frees the stack storage area occupied by argument values.
5. Execution continues in the calling program at the machine instruction after the jump-to-subroutine instruction.

### 19.2.2 Stack Diagrams and Program Traces

A **stack diagram** is a picture of the stack frames that exist at some given time during execution. We use stack diagrams to help understand what does and does not happen with function calls when we manually trace the execution of a program. They are particularly helpful in understanding recursion. Figure 19.1 shows a generic stack diagram. Figure 19.2 shows a more specific instance of a stack, which can be referred to during the following discussion.

To **trace** the operation of **a program**, start by drawing an area for global and `static` variables. Record the names and initial values of these variables. Set aside a second area for the program's output to simulate the screen. Begin the stack diagram at the bottom of the page with a frame for `main()` (new frames will be added above this one). Record the names and initial values of the variables of `main()` in this frame. Make three columns for each variable, giving its type, its name, and the initial value (or a ? if it is not initialized). Leave room to write in new values that might be assigned later.

Finally, start with the first line of code in `main()` and progress from line to line through the program code. Emulate whatever action each line calls for. If the line is an output command, write the output in your screen area. If the line is an assignment or input statement, change the value of the corresponding variable in the rightmost column of your stack diagram.

If the line is a function call, mark the position of the line in the calling program so that you can return to that spot. Add a frame to your stack diagram. Label the new frame with the function's name and record the argument values in it. Label these with the parameter names and add the local variables and their values.

These diagrams are pictures of the stack for the gas pressure program developed in Figure 9.5. The values shown for `temp`, `m`, and `vol` were entered by the user. On the left, the stack is shown during a call on the `ideal()` function, just before the function's `return` statement occurs. On the right, the stack is shown just before the return from `vander()`. The gray areas in each stack frame contain the function's return address and information used by the system for stack management.

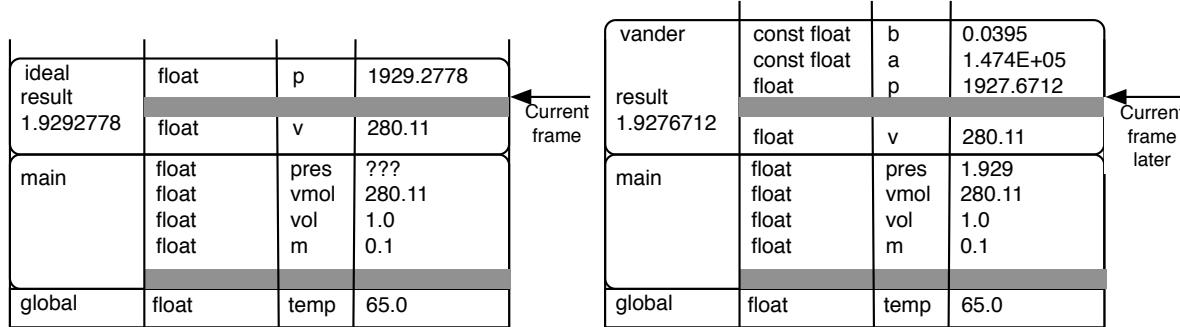


Figure 19.2. The run-time stack for the gas pressure program.

Then trace the code of the function until you come to the last line or a `return` statement. Write the function's return value under its name on the left side of the stack frame and draw a diagonal line through the function's stack frame to indicate that it has been freed.

For example, Figure 19.2 shows the stack at two times during the execution of the gas pressure program from Figure 9.5. The diagram on the left shows the stack during the process of returning from the `ideal()` function. In the next program step, the result will be stored in the variable `pres` of `main()` and the stack frame for `ideal()` will be freed. Later, `vander()` is called. The diagram on the right shows the stack when `vander()` is ready to return. The stack frame for `ideal()` is gone (it would be crossed out in your diagram), and a new stack frame for `vander()` is in its place.

## 19.3 Iteration and Recursion

In Chapter 5.8 we introduced a number guessing game that asked the human user to guess a secret number in the range 1...1000. If the user plays this game wisely, each guess will be halfway between the smallest and largest remaining possible values. By dividing the remaining interval in half each time and discarding one half, the secret number will eventually be the only possibility left. The player is able to halve the unexplored territory each time because, on each step, we tell the user which subinterval, left or right, contains the target number. This strategy is a form of **binary search**, which we formalize later in this chapter.

The binary search algorithm is an example of a general problem-solving approach called **divide and conquer**. If the problem is too difficult to solve when it is first presented, simplify it and try again. If this can be repeated, eventually the problem will become manageable. A loop, or **iteration**, is one way to repeat a process; recursion is another. We review the essentials of algorithms that iterate, then show how the problem simplification strategy is reflected in recursive processes.

### 19.3.1 The Nature of Iteration

Every loop consists of a control section and a body. The control section performs initialization and contains an expression, called the *loop condition*, that is tested before, during, or after every iteration. The loop ends when this condition has the value `false`. If we use a loop, something must change on each iteration (and that change must affect the loop condition) or the process will never end. We have seen infinite loops that fail to increment the loop variable or read a new input value; they simply repeat the same statements again and again until the user kills the process. Every programmer occasionally writes such a loop by mistake. In loops that are not controlled by the user's input, each iteration must make the remaining job smaller; that is, each trip around the loop must perform one step of a finite process, leaving one fewer step to do. Often progress is measured

by the value of a variable that is incremented or decremented on each iteration, until a predetermined goal is reached.

### 19.3.2 The Nature of Recursion

A *recursive function* does its task by calling upon itself, not by going around a loop. Like a loop, a recursive function must reduce the size of the remaining task on every call, or the process will become mired in an infinite repetition of the same process step; this is called **infinite recursion**. Eventually, an infinite recursion will be detected by the system and the process aborted with a comment such as **stack overflow**, due to the allocation of too many stack frames or too much memory.

When one function calls another, argument values are passed into the subprogram, local storage is allocated on the stack for them, and an answer eventually is passed back to the caller. The same is true when a recursive function calls itself. The stack provides storage for parameters and local variables for each call on a function. We say that a **call is active** until the called function returns and its frame is discarded. Normally, there are several active functions at once, because one function can call a second one, and the second can call a third, and so forth. Therefore, at any given time, many frames can be on the stack. The computer always is executing the most recently called function, so it uses the most recent frame. When the task of a function is complete, its stack frame is eliminated and it returns control to its caller. The caller, which had been waiting for the subprogram to finish, resumes its own task, using its own stack frame, which is now on top of the stack. A recursive function calls itself, so the caller and the subprogram are two activations of the same function.

If a recursive function calls itself five times, six stack frames will exist simultaneously for it, each holding the parameters for one of the active calls. Each time one of the calls returns, its stack frame is discarded and control goes back to the prior invocation. That is, when the fifth active call returns, control goes back to the fourth active call. The fourth incarnation then resumes its work at the spot just after the recursive call. This process proceeds smoothly, just as with ordinary function calls, so long as each call *specifies a smaller task* than the previous call. In theory, if the job does not get smaller, the recursions will continue endlessly. Practically speaking, though, a recursion is unlikely to last forever because each stack frame requires memory, thereby making the stack longer. Eventually, all of the available memory is used up and the stack becomes too large to fit into memory. At this time, the system will report that the stack has overflowed and abort the user's process.

As with loops, every recursion must contain a condition used to determine whether to continue with more recursive calls or return to its caller. This condition often is referred to as the **base case**, because the action to take (usually to return) is basic and straightforward. Frequently, the entire body of a recursive function is just a single **if...else** statement, where the **if** contains the termination condition, the **true** clause contains the **return** statement, and the **else** clause contains the recursive call. The simplest and most common recursive strategy is

- Base cases: The function checks the first base case and returns if it is satisfied. If needed, it checks the second, third, and further base cases and returns if any one of them is satisfied.
- Recursive step: If no base case is satisfied, the function calls itself recursively with arguments that exclude part of the original problem. When the recursive call returns, the function processes the answer, if necessary, and returns to the caller.

The program in Figure 19.3 applies this recursive strategy to the task of finding the minimum value in a list of data items. On each recursive step, the function processes the first item on the list and calls itself recursively to process the rest of the list. Thus, each step reduces the size of the task by one item. This pattern is typical of recursion applied to a list or array of data items, and is called a **simple list recursion**.

### 19.3.3 Tail Recursion

If the recursive call is the last action taken before the function returns, we say it is a **tail recursion**. Once the returns start in a tail recursion, all the calls return, one after another, and all the stack frames are discarded one by one until control reaches the original caller. The value computed during the last call is sent back through each successive stack frame and eventually returned to the original caller. The binary search function in Figure ?? is tail recursive; the results of the recursive calls are returned immediately without further processing. Any tail recursive algorithm can be implemented easily, and more efficiently, as a loop. The bisection program in Chapter 11 that finds the roots of an equation is a form of binary search implemented using a **do...while** loop.

However, not all recursive algorithms are tail recursive. For example, the `array_min()` function in Figure 19.3 is not tail recursive because the result of the recursive call is compared to another value before returning. Although we can easily write an iterative algorithm to find the minimum value in a list, many recursive algorithms cannot be rewritten easily as iterative programs; they are essentially recursive and have no simple, natural, iterative form. The last program in this chapter, quicksort, falls into this category. It is one of the most important and easily understood of these inherently recursive algorithms.

## 19.4 A Simple Example of Recursion

The program in Figure 19.3 uses recursion to find the minimum value in an array of values. Compare it to the iterative maximum-finding program in Figure 10.24. The iterative solution is better; it is just as simple and much more efficient. However, the recursive solution is useful for illustrating how recursion works and

---

```
#include <iomanip>
using namespace std;
#define N 15
int arrayMin( int a[], int n );
int main( void )
{
    int data[N] = { 19, 17, 2, 43, 47, 5, 37, 23, 3, 41, 29, 31, 7, 11, 13 };
    int answer;

    cout <<"\n Find minimum in an array of " <<N <<" positive integers.\n";
    answer = arrayMin( data, N );
    cout <<"\n Minimum = " <<answer <<"\n\n";
}

// -----
int arrayMin( int a[], int n )
{
    int tailmin, answer;
    cout <<" Entering: n=" <<setw(2) <<right <<n
          <<" head= " <<a[0] <<"\n";
    // Base case -----
    if (n == 1) {
        cout <<" ---- Base case ----\n";
        answer = a[0];
    }

    // Recursive step -----
    else {
        tailmin = arrayMin( &a[1], n-1 );
        if (a[0] < tailmin) answer = a[0];
        else answer = tailmin;
    }

    cout <<" Leaving: n=" <<setw(2) <<right <<n <<" head=" <<setw(2)
          <<a[0] <<" returning=" <<setw(2) <<answer <<"\n";
    return answer;
}
```

---

Figure 19.3. A recursive program to find a minimum.

developing some intuition for it. Later examples will demonstrate that recursive solutions can be easier to develop than iterative versions in some cases.

### Notes on Figure 19.3: A recursive program to find a minimum.

**Second box, Figure 19.3: the recursive descent.** We want to find the minimum value in a set of values. But that requires some work, so we use a strategy of shrinking the size of the problem. At each step, we defer the action of examining the first value in the set ( $a[0]$ ) and turn our attention to finding the minimum of the rest ( $a[1]$  to  $a[n-1]$ ). If we do this recursively, we put aside the values one at a time, until only one value remains. Once we reach a set of one, we know (trivially) that this value is the minimum, so we can simply return it to the caller. This occurs halfway through the entire process. At this point, we have progressed, recursively, to the end of the array, and reached the base case of the recursion. Given the sample `data` array, the program output up until this point is

```
Find minimum in an array of 15 positive integers.
Entering: n=15  head=19
Entering: n=14  head=17
Entering: n=13  head= 2
Entering: n=12  head=43
Entering: n=11  head=47
Entering: n=10  head= 5
Entering: n= 9  head=37
Entering: n= 8  head=23
Entering: n= 7  head= 3
Entering: n= 6  head=41
Entering: n= 5  head=29
Entering: n= 4  head=31
Entering: n= 3  head= 7
Entering: n= 2  head=11
---- Base case ----
```

Although we always print  $a[0]$ , it is not the same as `data[0]`, except on the very first call; it actually is the initial value of the portion of the array that is the argument for the current recursive call. During the processing, we repeatedly peel off one element and recurse. From the output, we can see that, after executing the first `printf()` statement 14 times, we have not yet reached the second `printf()` statement because of the recursions. Finally,  $n$  is reduced to 1 and we reach the base case.

**First box, Figure 19.3: the base case.** When writing a recursive function that operates on an array, the base case usually happens when one item is left in the array. In this example, given a set of one element, that element is the minimum value in the set. Therefore, we simply return it. The `true` clause of the `if` statement contains a diagnostic `printf()` statement; it is executed midway through the overall process, after the recursive descent ends and before the returns begin. If we eliminate this `printf()` statement, the base case becomes very simple:

```
if (n == 1)    return a[0];
```

The diagnostic output from the base-case call shows us that we have reached the end of the array:

```
Entering: n= 1  head=13
---- Base case ----
Leaving: n= 1  head=13  returning=13
```

Now we begin returning. The stack frame of the returning function is deallocated and the stack frame of the caller, which had been suspended, again becomes current. When execution resumes, we are back in the context from which the call was made. When we return from the base case, we go back to the context in which there were two values in the set.

**Second box revisited, Figure 19.3: ascent from the recursion.** At each step, we compare the value returned by the recursive call to  $a[0]$ . (Remember that  $a[0]$  is a different slot of the array on each recursion.) We return the smaller of these two values to the next level. Thus, at each level, we return with the minimum value of the tail end of the array and that tail grows by one element in length each time we return. When we

get all the way back to the original call, we will have the minimum value of the entire array, which is printed by `main()`:

```
---- Base case ----
Leaving: n= 1 head=13 returning=13
Leaving: n= 2 head=11 returning=11
Leaving: n= 3 head= 7 returning= 7
Leaving: n= 4 head=31 returning= 7
Leaving: n= 5 head=29 returning= 7
Leaving: n= 6 head=41 returning= 7
Leaving: n= 7 head= 3 returning= 3
Leaving: n= 8 head=23 returning= 3
Leaving: n= 9 head=37 returning= 3
Leaving: n=10 head= 5 returning= 3
Leaving: n=11 head=47 returning= 3
Leaving: n=12 head=43 returning= 3
Leaving: n=13 head= 2 returning= 2
Leaving: n=14 head=17 returning= 2
Leaving: n=15 head=19 returning= 2

Minimum = 2
```

## 19.5 A More Complex Example: Binary Search

Searching an arbitrary array for the position of a key element is a problem that has only one kind of general solution, the sequential search. If the data in the array are in some unknown order, we must compare the key to every item in the array before we know that the key value is not one of the array elements. Sequential search works well for applications like the Chauvenet table, presented in Chapter 10, where only a few values are looked through. For an extensive data set such as a telephone book or an electronic parts catalog, sequential search is painfully slow and completely inadequate. Therefore, large databases are organized or indexed so that rapid retrieval is possible. One such way to organize data is simply to sort it according to a key value and store the sorted data in an array. We have seen algorithms for the selection and insertion sort so far. The quicksort algorithm will be discussed in the next section.

Binary search is an algorithm that allows us to take advantage of the fact that the data in an array are sorted. (We simplify the following discussion by assuming that it is in ascending order. One of the exercises examines descending order.) The search can be implemented easily using either iteration or recursion. Both formulations are straightforward. The iterative version is a little more efficient, while the recursive version is a little shorter and easier to write. We will use the program in this section to illustrate both recursion and the binary search algorithm. The call chart in Figure 19.4 summarizes the overall structure of this program and where each piece can be found. Figures 19.5 through ?? contain a program that reads a sorted data file into an array, then searches for values requested by the user. We start with a `main()` program and its utility functions, then look at the recursive function last.

### Notes on Figure 19.4. Call chart for binary search.

In the diagram, the yellow box surrounds functions defined by the `Table` class. The blue-green boxes show functions from the system libraries `algorithm` and `vector`. The circular arrow represents the recursive call on `binSearch()`.

### Notes on Figure 19.5. The binary search main program.

#### *First box, Figure 19.3: The file name.*

- This program asks the user to enter the name of the data file.
- The function `getline()` reads an input of indefinite length and stores it in a C++ string variable. The string variable will grow as long as necessary to contain all the input up to a newline.

#### *Second box, Figure 19.5: Entering the OO world.*

- A always, the job of `main` is to instantiate one `Table` object and call its primary function to search the data for key values entered by the user.

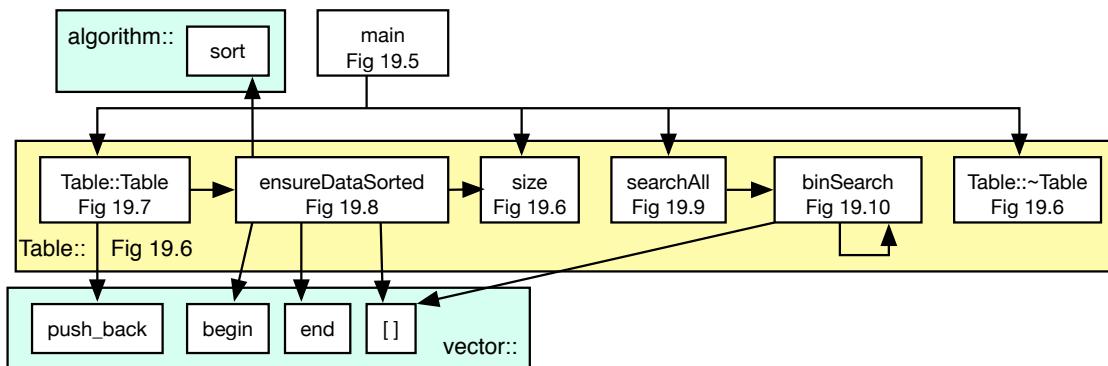


Figure 19.4. Call chart for binary search.

- We create the `Table` object to use the filename just entered. The constructor will open and read the file, and store the contents in a `vector`. The vector will grow as long as necessary to store all the data in the file.
- It is always a good idea for `main()` to display information for the user at the end of every major step of its process.
- Finally, now that the data is ready to use, we call `searchAll`, the primary function in the `Table` class.

*Sample output from main():*

```

Find a key value in an array of integers, 0 ... 1000.
Enter name of data file to be searched: binsrch.dat.txt
17 data items read; ready to search.
  
```

**Notes on Figure 19.6. The Table class.**

*First box, Figure 19.6: The vector.*

- A `vector` is a “smart” array: it starts empty, at a default length, and grows repeatedly by doubling its length each time. Growth happens when the current array is full and your program tries to put another data item into it. To use `vector`, `#include <vector>`.
- The self-adjusting nature of a `vector` is especially useful for input, if you do not know how much data to expect.

This `main()` program uses the class defined in Figure 19.6 and Figure 19.7

```

#include "table.hpp"
// -----
int main( void )
{
    string inname;

    cout << "\n Find a key value in an array of integers, 0 ... 1000.\n"
        << " Enter name of data file to be searched: ";
    getline( cin, inname );

    Table t( inname );
    cout << " " <<t.size() << " data items read; ready to search.\n";
    t.searchAll();

    return 0;
}
  
```

Figure 19.5. The binary search main program.

---

noindent This class is instantiated by the `main()` program in Figure 19.5.

```
#include "tools.hpp"
#define NOT_FOUND -1

class Table {
private:
    vector<int> data;

    void ensureDataSorted();
    int binSearch( int left, int right, int key );

public:
    Table( string inname );
    ~Table(){}
    void searchAll(); int size(){ return data.size(); }

};

};
```

---

**Figure 19.6.** The Table class declaration.

**Second box, Figure 19.6: Private functions.**

Neither one of these functions is ever called from outside the class.

- `ensureDataSorted()` is called from the Table constructor if the input file was not sorted.
- `binSearch()` is the recursive search function, called from `searchAll()` to search for a series of inputs in the sorted file.

**Third box, Figure 19.6: Public functions.**

These four functions are all called from `main()`.

- The Table constructor sets up the sorted data array, ready to search.
- `searchAll()` does the actual search.
- The Table destructor is called implicitly from `main()` when the program finishes. Note that it does nothing, because the Table functions did not create any dynamic memory.
- The destructor and the `size()` function are both short inline functions. An inline function is more efficient at run-time than an ordinary function.
- `size()` returns the answer from calling `vector::size()`. “Wrapper” functions like this are often used with data structures defined by the C++ libraries.

**Notes on Figure 19.7: The Table constructor.**

This program uses data from the keyboard to open a data file. The data is supposed to be in ascending sorted order. Therefore, reading the input really is a three-step process: (1) identify and open the file, (2) read the information, and (3) verify that it is sorted in ascending order.

**First box, Figure 19.7: Opening the input stream.**

- The data file name was read in `main()` and has been passed to the constructor as a parameter. The first line opens the stream.
- It is always necessary to check whether a file is properly open. In this case, we abort if it is not.
- The function `fatal()` is used to print an error comment. The The first argument is a C-style format. The second is a c-string, as required by the `%s` in the format,
- We can extract the C-string from the C++ string by using the `.data()` function or the `.c_str()` function.

```

Table:: Table( string inname ) {
    int input;
    ifstream infile( inname );
    if (!infile.is_open()) fatal( " Cannot open input file %s", inname.data() );

    for( ;; ){
        infile >> input;
        if( ! infile.good() ) break;
        data.push_back( input );
    }

    ensureDataSorted();
}

```

**Figure 19.7.** The Table constructor.

- Here is the error report that resulted from entering an incorrect file name:

```

Find a key value in an array of integers, 0 ... 1000.
Enter name of data file to be searched: bins.txt
Cannot open input file: bins.txt
Error exit; aborting.

```

*Second box, Figure 19.7: Reading the data.*

- When using `>>` to do input, a program must test for end-of-file *after* reading some data, not *before*. So we use the indefinite for loop, not a while loop.
- Inside the for loop there is an `if...break` that tests for end-of-file and ends the loop when that happens.
- The `if...break` will also end the loop if there was a hardware read error or the file contains non-numeric data.
- Once read, each input number is stored in the next available slot of the vector named `data`.

*Third box, Figure 19.7: A precondition.*

- A binary search program cannot function properly unless the data in the array are sorted in the correct order, in this case, ascending<sup>3</sup> order.
- Rather than take it on faith that the numbers are sorted, we call the function in Figure 19.8 to check the order and, if an error is found, sort the data.

**Notes on Figure 19.8: Ensuring that the data is sorted.** One of the principles of OO design is that every class should take care of its own emergencies. In this case, it means that we should make sure the data in the array is sorted before trying to use binary search on it.

*Outer box, Figure 19.8: Check the whole array.*

Note that this loop starts by comparing the contents of slots 0 and 1. We cannot use a for-each loop here because only `size - 1` comparisons are required to sort `size` numbers.

**Middle box, Figure 19.8: Test the order of two items.** If two adjacent items are in ascending order, all is well and there is nothing to do. That is why there is no `else` clause for the `if`.

**Inner box, Figure 19.8: Sort if necessary.** If any two adjacent items are not in ascending order, the array is not sorted. In that case, we call the `sort()` function from the algorithms library. You can do this with any vector. `.begin()` and `.end()` are defined as iterators that point to the first thing in the vector and the first address that is past the end of the data in the vector.

---

<sup>3</sup>Technically, we require non-descending order.

---

This function is called from the Table constructor in Figure 19.7

```
void Table::  
ensureDataSorted() {  
    for (int k = 1; k < data.size(); ++k) {  
        if (data[k-1] > data[k]) {  
            sort( data.begin(), data.end() );  
            return;  
        }  
    }  
}
```

---

**Figure 19.8.** Ensuring that the data is sorted.

**Notes on Figure 19.9: Interaction with the user.**

It is often not necessary to give a full prompt for every user input. In this case, we prompt at the beginning and not every time around the input loop.

**First box, Figure 19.9: Prompt once for all.**

The input variable, `key` is declared to be an `int`. Therefore, when you write `cin >> key`, the system expects the user to type in a number.

**Second box, Figure 19.9: Search for another?**

- The prompt at the top tells the user to enter numbers (plural) and to enter a ‘.’ to quit. Actually, since we are doing integer input, *any* non-digit will cause a stream error (faulty input conversion).
- Thus, you could enter a period, as instructed, or enter a letter or special character .... Any of them will cause the stream to set an error flag.
- The third line in this box tests for a stream error. The same test can be used to find an end-of-file, but here we are actually looking for a conversion error.

---

This function is called from the main program in Figure 19.5

```
void Table::  
searchAll() {  
    int slot, begin, end, key;  
    cout << "\ Enter numbers to search for; period to quit.\n";  
    for(;;) {  
        cout << " What number do you wish to find?  ";  
        cin >> key;  
        if (!cin.good()) break;  
  
        cout << "  " <<key << endl;  
  
        slot = binSearch( 0, data.size(), key );  
        if (slot == NOT_FOUND) cout << " is not in table.\n\n";  
        else cout << " was found in slot " <<slot << ".\n\n";  
    }  
}
```

---

**Figure 19.9.** Searching for numbers.

---

This recursive function is called from `searchAll()` in Figure 19.9. Diagrams of the operation start in Figure 19.11.

```
int Table::  
binSearch( int left, int right, int key ) {  
    int mid = left + (right-left)/2;      // Compute middle of search interval  
    cout << " left=" <<left << " mid=" <<mid << " right=" <<right << endl;  
  
    // Base cases: (1) we found it or (2) it's not there  
    if (left >= right) return NOT_FOUND;  
    if (data[mid] == key) return mid;  
  
    // Recursion: search a smaller array  
    if (key < data[mid]) return binSearch( left, mid, key );  
    else                  return binSearch( mid + 1, right, key );  
}
```

---

**Figure 19.10. The binary search function.**

**Third box, Figure 19.9: Debugging technique.**

When trying to debug a loop or a recursion it is always a good idea to put a printout in your code that will help you track the progress of the algorithm. This debugging printout echoes the user's input, a necessary step to be sure that the data is being read correctly.

**Fourth box, Figure 19.9: The search.**

- The actual recursive search is done in `binSearch()`. This function sets up the search, receives the found/not found answer, and displays it for the user.
- In the first line, we call `binSearch()`. The third parameter is the number we are searching for.
- The first and second subscripts are integer subscripts because we want to do arithmetic with them: 0 is the subscript of the first data item in the array.
- `data.size()` is the number of data values in the array and also an *offboard* subscript for the end of the data. That is, `data.size()` is the subscript for the first slot that does not contain data.

**Notes on Figure 19.10: The binary search function.**

We search for the key value among the  $n$  values in the data array, which are sorted in ascending order.

**First box, Figure 19.10: Split the array.**

- On each recursion, the left and right subscripts are closer to each other. To find the subscript of the middle of the array, we use `right - left` to calculate how far apart they are, then add half the distance to `left` to get the middle.
- This calculation uses integer arithmetic. If there are an odd number of things, the answer is rounded down.
- Debugging a recursive function can be a challenge unless the programmer has enough information to track the progress of the recursion. To find and eliminate an error, the programmer needs to know the actual arguments in each recursive call. For this purpose, while debugging this program, the programmer inserted an output statement in this function just after the initialization of `mid`.

```
int mid = left + (right-left)/2;  
cout << " left=" <<left << " mid=" <<mid << " right=" <<right << "\n";
```

**Second box, Figure 19.10: Check the base cases.**

There are two base cases:

- We can guarantee that the recursion will eventually end by testing and eliminating the item at the split point on every call. This guarantees that each successive recursive call will be searching a shorter list of possibilities.
- If the left and right subscripts have met and passed each other in the middle, the key value is not in the array and we return an error code. The symbolic name for this code was defined in the Table header file.
- If the value at `mid` matches the key value, we return the subscript at which the key was found.
- The order of these two comparisons *does* matter. It is possible for `left`, `right`, and `mid` to be all the same subscript. This happens when the remaining unsearched portion of the array is empty, but also if the array was empty when the search began. If there is no data at all in the array, or if `left`, `right`, and `mid` are all offboard (past the end of the data), it is an error to test the data at `mid`. So we must test whether data exists first.

**Third box, Figure 19.10: Recurse.**

- First, we test whether the key is in the bottom or the top half of the remaining area. Then we recurse on one of the halves, never both. We already checked the `mid` value, so that is excluded from both recursions.

**Output from the binary search test run with debugging printouts:**

```
Enter numbers to search for; period to quit.
```

```
What number do you wish to find? 345
```

```
345
```

```
left=0  mid=8  right=17
left=0  mid=4  right=8
left=5  mid=6  right=8
left=5  mid=5  right=6
left=6  mid=6  right=6
is not in table.
```

```
What number do you wish to find? 387
```

```
387
```

```
left=0  mid=8  right=17
left=0  mid=4  right=8
left=5  mid=6  right=8
was found in slot 6.
```

```
What number do you wish to find? 0
```

```
0
```

```
left=0  mid=8  right=17
left=0  mid=4  right=8
left=0  mid=2  right=4
left=0  mid=1  right=2
left=0  mid=0  right=1
left=0  mid=0  right=0
is not in table.
```

```
What number do you wish to find? 967
```

```
967
```

```
left=0  mid=8  right=17
left=9  mid=13  right=17
left=14  mid=15  right=17
left=16  mid=16  right=17
was found in slot 16.
```

**Output from the binary search test run without debugging printouts:**

```
Enter numbers to search for; period to quit.
```

```
What number do you wish to find? 345
```

```
345 is not in table.
```

We are prepared to begin a binary search of the array `data`, shown below, looking for the key value 270. The `left`, `right`, and `mid` subscripts are shown as they would be after making the first call at the beginning of this search.

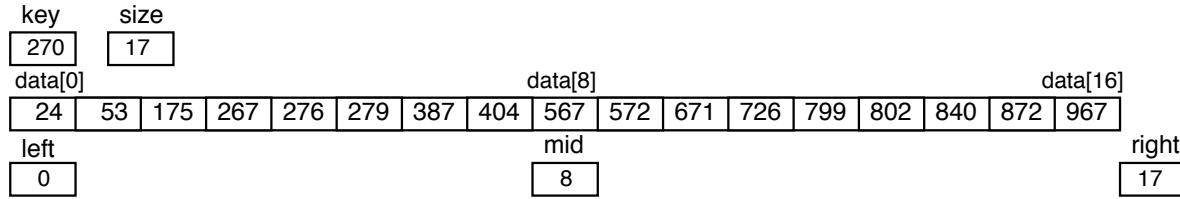


Figure 19.11. Diagrams of a Binary Search: Ready to begin.

What number do you wish to find? 387  
387 was found in slot 6.

What number do you wish to find? .

#### Notes on Figure 19.11: Ready to begin.

- Initially `left` = 0, `right` = the number of items in the array, and `mid` is halfway between them.
- Note that all the data slots are colored white. They will become gray when they have been eliminated from scope of the the remaining search.

#### Notes on Figure 19.12: After the first test.

- After comparing the key value 270 to the value of `data[mid]` = 567, the program knows that the key should be in the left half.
- `bin_search()` is called recursively to search an interval with the same left end but with `right` = `mid`, since `mid` has been checked and `right` is supposed to be offboard.

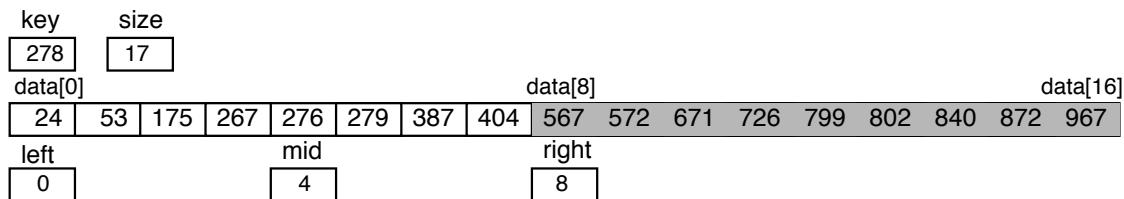


Figure 19.12. Second active call.

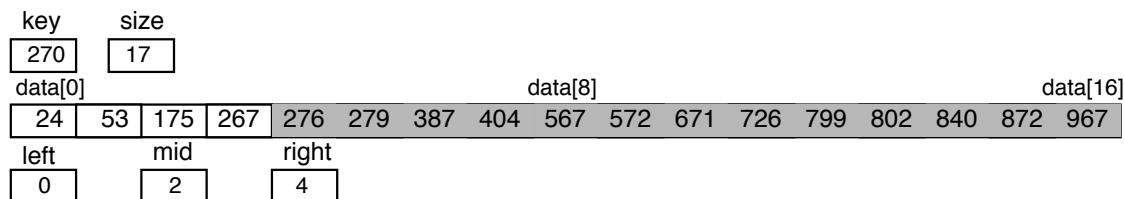


Figure 19.13. Third active call.

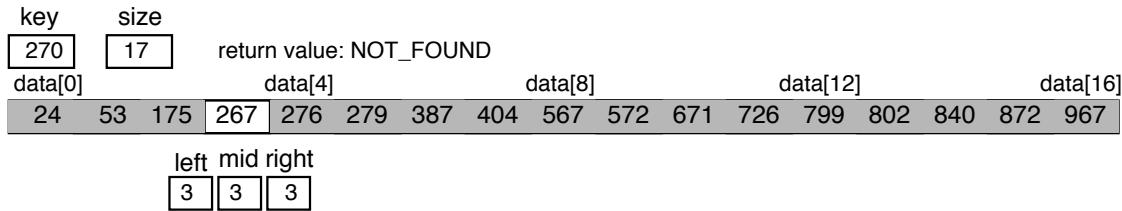


Figure 19.14. Fourth active call.

**Notes on Figure 19.13: After the second test.**

- After comparing the key value 270 to the value of `data[mid]` = 276, the program calls `bin_search()` recursively to search an interval with the same left end but with `right` = 4.

**Notes on Figure 19.14: After the third test.**

- On this call, the left end of the remaining interval has been eliminated. The new left end is `mid+1` = 3 and `right` is still 4, so there is one thing left to search.

The key is 270 does not match the value in that slot 3, so the program returns the failure code.

## 19.6 Quicksort

Even though computers become faster and more powerful each year, efficiency still matters when dealing with large quantities of data. This is especially true when large data sets must be sorted. Some sorting algorithms (selection sort, insertion sort) take an amount of time proportional to the square of the number of items being sorted and so are only useful on very short arrays (say, 10 items or fewer).<sup>4</sup> Even the fastest computer will take a long time to sort 10,000 items by one of these methods. The fastest known sorting algorithm, **radix sort**, takes an amount of time that is directly proportional to the number of items, but it requires considerable effort

<sup>4</sup>The theory behind this is beyond the scope of this book.

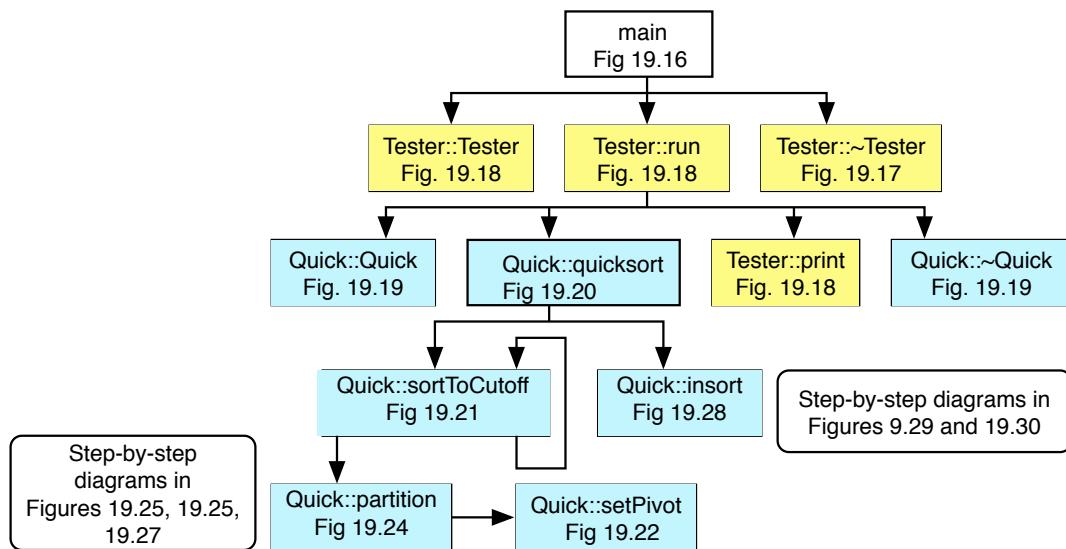


Figure 19.15. Call graph for the quicksort program.

---

This `main()` program uses the `Tester` class in Figures 19.17 and 19.18.

```
#include "tools.hpp"
#include "Tester.hpp"

#define LENGTH 1000000

// -----
int main(void) {
    banner();
    Tester( LENGTH ).run();
    bye();
}
```

---

**Figure 19.16.** The main program for `quicksort()`.

to set up the necessary data structures before sorting starts. Therefore, it is useful only when sorting very long arrays of data. In contrast, the `quicksort` algorithm is one of the fastest ways to sort an array containing a moderate number of data items (up to millions).

Quicksort is an example of the strategy of divide and conquer: Divide the problem repeatedly into smaller, simpler tasks until each portion is easy and quick to solve (conquer). If this can be done in such a way that the collection of small problems actually is equivalent to the original one, then after solving all of the small problems, the solution to the large one can be found. In particular, the quicksort algorithm works by splitting an array of items into two portions so that everything in the leftmost portion is smaller than everything in the rightmost portion, then it recursively works on each section. At each recursive step, the remaining part of the array is split again. This splitting continues until each segment of the array is very small (often one or two elements) and can be sorted directly. After all of the small segments have been sorted, the entire array turns out to be sorted as well, because every step put smaller things to the left of larger things.

There have been many implementations of quicksort, some very fast and others not so fast. In general, if an array of items is in random order, any of the quicksort implementations will sort it faster than a simple sort (selection, insertion, etc.). The version of quicksort presented here is especially efficient and one of the best, because its innermost loops are very short and simple. Further optimizations that can improve the efficiency of this algorithm have been omitted from the code in this section for the sake of clarity. However, two possible improvements are mentioned at the end of the section and explored in the exercises.

As an aid to following this example, a call graph is given in Figure 19.15. It includes references to the figures in which each function is defined or its actions are illustrated. We begin the discussion with the `main()` program in Figure 19.16, which calls the `quicksort()` function in Figure 19.20.

#### Notes on Figure 19.16: The main program for `quicksort()`.

This is a typical C++ main program: it creates a `Tester` object and calls its `run` function. The calls on `banner()` and `bye()` let the user track the progress.

The `#define` at the top makes it easy for us to test the program with any size data set. This is a very fast implementation of the algorithm. Given the amazing speed of today's computers, we need to sort a million numbers to even notice or measure the execution time.

#### Notes on Figure 19.17. Tester generates data for `quicksort()`.

##### *First box, Figure 19.17: data members.*

We need only two data members: a dynamic array and the length of that array. The array is allocated dynamically so that we can test the quicksort with data sets of varied sizes. The length of the array is `#defined` at the top of `main` and passed as a parameter to the `Tester` constructor.

##### *Second box, Figure 19.17: Create the data, sort it, and output it.*

- The `Tester` constructor generates random data for the test. It allocates a long dynamic array for the data, so it needs to receive the desired array length from `main()`. See the first box in Figure 19.18

---

This function is called from the `main()` program in Figure 19.16. It generates the required amount of random data and stores it in the data array to be sorted. Then it creates a Quick object to do the sorting.

```
#pragma once
class Tester {
private:
    int length;
    int* data;
public:
    Tester( int len );
    ~Tester() { delete [] data; }
    void run();
    void print() const;
};
}
```

---

**Figure 19.17.** Tester generates data for `quicksort()`.

- The `Tester` destructor must deallocate the dynamic array. That is a simple one-line task, so the function is declared inline in the .hpp file.
- The `run()` function calls the `quicksort` function. See the second box in Figure 19.18
- The `print()` function outputs the results to a file, not to the screen, because we expect the results to be voluminous. It is declared as a `const` function because a `print()` function should never modify its object. See the third box in Figure 19.18

#### Notes on Figure 19.18. Implementation for the Tester class.

##### *First box, Figure 19.18: the constructor.*

- The `Tester` constructor uses ctors (constructor initializers) to initialize the two data members of the new object. The ctor list starts with a colon after the parameter list of the constructor and before the open brace at the beginning of the body. The ctors are always executed before the body of the method.
- The first ctor, `length(len)` stores the parameter named `len` into the data member named `length`.
- The second ctor, `data(new int[len])` allocates a long dynamic array for the data, and stores the pointer into the class member named `data`.
- The body of the constructor is a loop that generates random numbers and stores them in the data array.

##### *Second box, Figure 19.18: Test the quicksort.* The `run()` function prints user information, calls the `quicksort()` function, then writes the sorted data to a file by calling `print()`.

##### *Third box, Figure 19.18: Print to a file.*

- Lines 1, 2, and 4 of the `print()` function are needed to open a file for the output, verify it, and close it.
- Line 3 is a simple loop to do the output. Note that is is very much like the loop that generated the data.

#### Notes on Figure 19.19: the Quick class.

##### *First box, Figure 19.19: the data members.* To sort, we need to know where the data begins and how much data we have. These are passed into the constructor by the `Tester::run()` and stored in member variables. They are accessed by every function in the class.

*Third box, Figure 19.19: the public class interface.*

- There are only two public functions: the `Quick` constructor and `quicksort()`. All other work is done by private helper functions.
- The constructor is an inline function with no body. Its uses ctors to initialize the two data members. The ctors must be given in the same order as the definitions of the class members.

*Second box, Figure 19.19: the private functions.*

- `setPivot()`: On each recursion, a quicksort algorithm chooses a pivot value and puts it in place in the array.
- `partition()`: Then the `partition()` function separates the large and small data items. Items that are smaller than the pivot are moved to the low-subscript end of the array. Items that are larger than the pivot are moved to the high-subscript end. Items equal to the pivot might end up in either portion. After examining all array elements, and moving them to the right array area, the pivot value is placed between the two areas.
- `sortToCutoff()` is the actual recursive sorting function. The recursion ends when the number of data items remaining to be sorted no longer justifies the overhead inherent in doing recursion. This leaves the data items almost, but not quite, sorted.

This function is instantiated by the `main()` program in Figure 19.16. It generates the required amount of random data and stores it in the data array to be sorted. Then it creates a `Quick` object to do the sorting.

```
#include "Tester.hpp"
#include "quick.hpp"
#include "tools.hpp"

// Allocate and fill data array with random non-negative integers.

Tester:::
Tester( int len ) : length(len), data(new int[len]) {
    for (int k=0; k<length; k++) data[k] = rand();
}

// Run test of quicksort on random data array -----
void Tester:::
run() {
    cout << "\ Quicksort program, cutoff=" << CUTOFF << ".";
    cout << " " << length << " data items generated; ready to sort.\n";
    Quick( length, data ).quicksort();
    cout << " Data sorted; writing to output stream.\";
    print();
}

// Print array values to selected stream. -----
void Tester:::
print() const {
    ofstream out("sorted.txt");
    if(!out.is_open()) fatal( "Cannot open output stream %s", "sorted.txt" );
    for (int k=0; k<length; k++) out << data[k] << endl;
    out.close();
}
```

Figure 19.18. Implementation for the `Tester` class.

---

```

#pragma once
#include "tools.hpp"

#define CUTOFF 20

typedef int BT; class Quick {
private:
    BT* data; // Array containing data to be sorted.
    int n; // Number of items to be sorted, initially

    // Prototypes of private, non-inline functions.
    void insert(BT* begin, BT* end);
    BT setPivot(BT* begin, BT* end);
    BT* partition(BT* begin, BT* end);
    void sortToCutoff(BT* begin, BT* end);

public:
    Quick( int n, BT* d ) : data(d), n(n) {}
    void quicksort();
};


```

---

**Figure 19.19.** The Quick class.

- `insert()` is an insertion sort that is used after the cutoff to adjust the positions of the almost-sorted items.

**Notes Figure 19.20: The quicksort() function.**

This function is called from the `Tester::run()` function in Figure 19.18. In turn, it calls the functions in Figure 19.21 and 19.28.

- The quicksort algorithm is much faster than any double-loop sorting algorithm if the number of things to be sorted is large. However, insertion sort is faster if there are only a few things to sort.
- In the code, there are two function calls. The first calls a recursive function, `sortToCutoff()` to subdivide the data set into a series of 1 to CUTOFF chunks. When it finishes, all of the data values in each chunk are smaller than all the data values in the next chunk. That is, the array is mostly sorted.
- Then `insert` (an ordinary insertion sort) is called to finish sorting the items within each chunk. A CUTOFF of 20 to 25 was determined experimentally. Machine architecture and OS cacheing strategies make this larger or smaller on different systems.

**Notes Figure 19.21: sortToCutoff().**

*First box, Figure 19.21: sortToCutoff()*

- This function is called with a pointer to the beginning and end of the portion of the array that will be sorted.

---

```

#include "quick.hpp"

void Quick::
quicksort() {
    sortToCutoff(&data[0], &data[n]); // Recursively sort down to CUTOFF-sized blocks.
    insert(&data[0], &data[n]); // Use insertion sort to finish the sorting
}


```

---

**Figure 19.20.** The quicksort() function.

---

```

// Recursively sort down to CUTOFF-sized blocks. -----
void Quick::
sortToCutoff(BT* begin, BT* end) {
    if (end - begin > CUTOFF) {
        BT* split = partition( begin, end );
        sortToCutoff( begin, split );
        sortToCutoff( split+1, end );
    }
}

// else there is nothing more to do. Just return.
}

```

---

**Figure 19.21.** `sortToCutoff ()`

- Subtracting the beginning pointer from the end pointer uses pointer arithmetic. The result is the number of items in this part of the array. If it is  $>$  the CUTOFF, processing continues, otherwise the call returns without doing anything and the recursion ends.
- If there are still  $>$  CUTOFF items to sort, we call the `partition()` function to divide the data into two blocks.
- When `partition()` finishes, all the data in the left-hand block will be  $\leq pivot$ , all the data in the right-hand block will be  $\geq pivot$ , and the pivot value will be stored in the array between the two blocks.
- The pivot value is in its final, sorted, position, and the remaining data items are split into two blocks that can be sorted independently and will not be mixed or merged later.
- At this point we call `sortToCutoff()` recursively, twice, to sort the lower and upper blocks.

#### Notes on Figure ???: Setting the pivot.

There are many implementations of the general quicksort strategy – some are very fast, some are much slower. This implementation is one of the fastest. The most important improvements are in the `setPivot()` and `partition()` functions.

#### *First box, Figure ???: Identify 3 candidates for being the pivot.*

- We want the pivot to be the median of 3 data values, chosen from different parts of the data array. This helps to avoid worst-case behavior.

---

This function is in the file `quick.cpp`. It is called from `sortToCutoff()` in Figure 19.21.

```

#include "quick.hpp"
//-----
// Choose a pivot value, use the median of the first, last, and middle values.
// Place the median at the beginning of the array (*begin).
// Put the maximum value in *last and the minimum in *mid.

BT Quick::
setPivot(BT* begin, BT* end) {
    BT* last = end - 1;                                // last is onboard, end is offboard.
    BT* mid = begin + (last - begin) / 2;              // Find the middle.

    if (*mid > *last) swap(*mid, *last); // Swap larger into end position.
    if (*begin > *last) swap(*begin, *last); // Maximum item is now in place.
    if (*begin < *mid) swap(*begin, *mid); // Put median in slot at beginning.

    return *begin;
}

```

---

**Figure 19.22.** Setting the pivot.

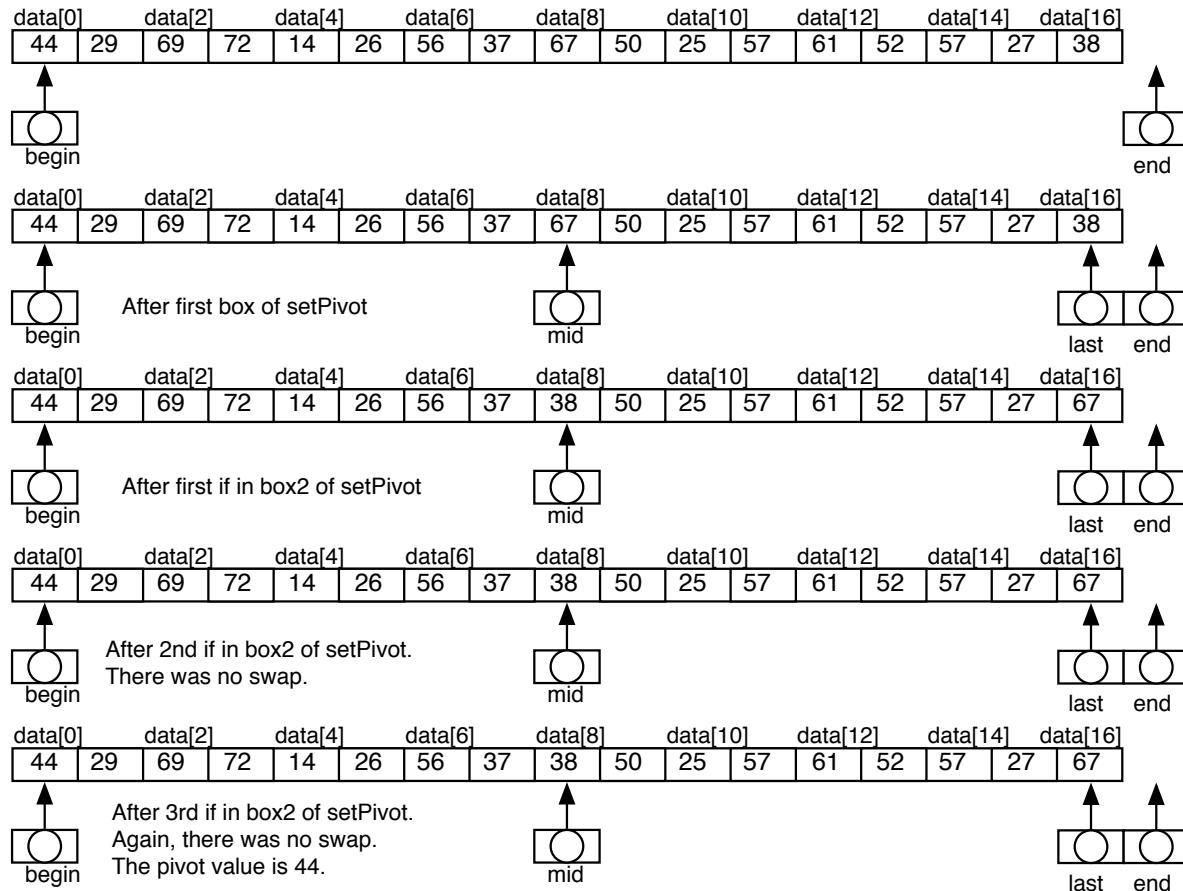


Figure 19.23. A diagram of `setPivot()`.

- For two of the three data values, we use the first and last in the array (`*begin` and `*last`). Using the first and last values avoids worst case behavior when the data is nearly sorted but ends with some unsorted values.
- The third value is taken from the middle of the array (`*mid`). If the array is nearly sorted, this value should approximate the actual median value.

#### *Second box, Figure ?? set up sentinel values for the partitioning scans.*

- During the partition step, we want the pivot to be at the beginning of the part of the array we are sorting.
- We use three `if` statements to compare `*begin`, `*mid`, and `*last` to find the median. In the process, we also move the largest to the right end of the array and move the pivot to the left end.
- After the second `if` statement, we know the largest value of the three is in the last array slot. The third `if` compares the other two values and puts the median value into slot `begin` to be used as the pivot.
- The pivot and the maximum value are now at the two ends of the array. These two values act as sentinels when the array is scanned. They ensure that the scanners do not “fall off” the ends of the array.
- Without a sentinel, the inner loop must compare two data items *and* check whether it has reached the end of the array. Using a sentinel makes the second check unnecessary and significantly speeds up the loop.
- The sentinel on the right end of the array can be any value  $\geq \text{pivot}$ , because this will end the comparison loop for the scanner named `luke`, that starts at the left and moves to the right. (Luke likes little things and throws the big things out of his area.)

```

#include "quick.hpp"

// Move smaller items to left end of the array, larger ones to the right end.
BT* Quick::
partition(BT* begin, BT* end) { // end is an off-board end pointer.

    BT* luke = begin;           // Luke likes little things.
    BT* rose = end - 1;         // Rose starts on-board at the end.
                                // Rose likes big, robust things.
    BT pivot = setPivot(luke, rose); // Now pivot element is in *begin

    for (;;) { // Loop until Luke meets Rose somewhere in middle of array.
        while (*++luke < pivot); // stops at begin bith thing, or *last
        while (*--rose > pivot); // stops at 1st little thing, or *begin

        if (luke >= rose) break; // L and R have met in the middle.
        swap (*rose, *luke);    // Little goes on left, big on right.
    }

    // Now, left of rose all is < pivot and right of Rose, all is > pivot.
    *begin = *rose, *rose = pivot; // So swap pivot to rose's position.
    return rose;                // Return subscript of split-point.
}

```

Figure 19.24. The `partition()` function, the heart of `quicksort()`.

- The sentinel on the left end of the array is a copy of the pivot value, because this will end the comparison loop for the scanner named `rose`, that starts at the right and moves to the left. (Rose relishes robust things and throws the little things out of her area.)

*Third box, Figure ???: returning the pivot.*

This function is called from `partition` and returns the value of the pivot to `partition`, where it is stored in a local variable named `pivot`.

**Notes on Figure 19.23: The `setPivot()` function.**

In this diagram, we walk through each step of the `setPivot()` function the first time `quicksort()` is called on the array shown.

- First line: note that `end` is an off-board pointer and the last data is at slot `end-1`.
- Second line: using integer arithmetic,  $(17 - 0)/2 = 8$  so slot 8 is the middle of the array.
- Now we use three compare-then-swap instructions to find the median of three numbers, 44, 67, and 38.
- Third line: We compare the mid and last values (67 and 38) and put the larger (67) in the last slot.
- Fourth line: Now we compare the first and last values (44 and 67). Since the larger value (67) is already in the last slot, no swap is needed. This happens half of the time.
- Now we know that the last slot contains the maximum, but we do not yet know whether the minimum is in slot `begin` or the middle slot.
- Fifth line: So we compare those values and swap, if necessary, to put the median value in slot `begin`. In this case, no swap is needed.
- The value now at the beginning of the array will be the pivot value for the partition step.

**Notes on Figure 19.24: The `partition()` function, the heart of `quicksort()`.**

Most implementations of partition use the variable names `i` and `j`. However, we use the variable names `luke` and `rose` because the reader is less likely to get mixed up about which scanner is moving in which direction.

*First box, Figure 19.24: set the pointers and the pivot.*

- Most of the execution time of `quicksort()` is spent in the `partition()` function. This implementation is very fast because the two partitioning loops do very little work.
- To speed execution, we use pointers instead of subscripts. We set pointers named `luke` and `rose` to slot `begin` of the array and to the last element in the array.
- After calling `setPivot()`, both of these array positions will contain sentinel values that have already been checked during the pivot computation. Note that this call also returns the pivot value, which is stored in the variable `pivot`.
- So `pivot` holds a copy of the value in slot `begin` of the array. Using the copy speeds up the comparisons during the inner scan loops because no dereference or subscript operation is needed.

*Second box, Figure 19.24: scanning.*

- The heart of a quicksort is the **partition step**, which splits the array into two parts, separated by one item at the split point. The split is done in such a way that everything in the left section is smaller than or equal to the value at the split point, and the split value is smaller than or equal to everything in the right section.
- In most implementations, the task is performed by the function named `partition()` (Figure 19.24). The value returned by `partition()` is a pointer to the split position. This state is shown in the example in Figure 19.22. This slot is gray to indicate that the value stored there is in its final, sorted position.
- Before every iteration, both scanners are pointing at values that have already been compared. So the first action is to increment or decrement the scanner.
- The two scanning loops in the second box have no loop body. The increments that are part of the while-test do all the necessary work. Because there are sentinels on the ends of the array, we do not need to test for the end of the array: the sentinel will stop the loop.
- When `luke` stops scanning, he is pointing at a data that is too big to be in his part of the array. When `rose` stops, she is pointing at a data value that is too small for her.

*Third box, Figure 19.24: partitioning.*

- When control gets to the third box, there are two possibilities and two possible next actions.
  - Luke is still to the left of rose, so they have not yet looked at all the data in the array. They swap data items and continue with the scan.
  - They have crossed, or are sitting on the same array slot. This means that they have examined all the data in the array and they are done. So they break out of the outer loop.

*Fourth box, Figure 19.24: put the pivot in place.*

- When `luke` and `rose` cross, `rose` is pointing at an item that is acceptable on the left (small value) end of the array.
- This value is swapped with the pivot value, which is in slot `begin`.
- Now the pivot value is in its final resting place and the data items have been partitioned into smaller and larger sets. Partition returns `rose`, a pointer to the split-point. Then `sortToCutoff()` uses that pointer to calculate parameters for the next pair of recursions.
- If the data set has several copies of the pivot value, some copies are likely to end up in both halves of the array. That is OK. The algorithm still works.
- If the scan stops with `luke` and `rose` pointing at the same slot, that slot contains a copy of the pivot. That is OK.
- Every recursive call on `sortToCutoff()` starts with one big block of data items and separates them into two smaller blocks separated by a sorted pivot value. The most desirable division, of course, is an even split, which rarely happens with random data. That is OK.
- It is possible for the split point to be anywhere in the array interval, except for slot `begin` and the last slot. The split point cannot be there because we put a value  $\geq$ pivot at the right end, and we know that a value  $\leq$ pivot will end up at the right end. That is OK. This might cause an additional recursion, but the algorithm will still work.

- When the last recursive call on `sortToCutoff()` returns, the data is approximately sorted. One trip through insertion sort fixes the little problems that remain.

**Notes Figure 19.28: Finish sorting using insertion sort.**

This function is called by `quicksort()`, Figure 19.20.

- We improve the efficiency of quicksort by terminating the recursions before the data is fully sorted, when the number of items left to sort is small enough so that insertion sort is faster. This is because every recursive call builds (and later tears down) a stack frame, while loops simply increment a counter in the current stack frame.
- The best cutoff may be different for different platforms. For old single processor machines, the threshold was 8 to 10 items.

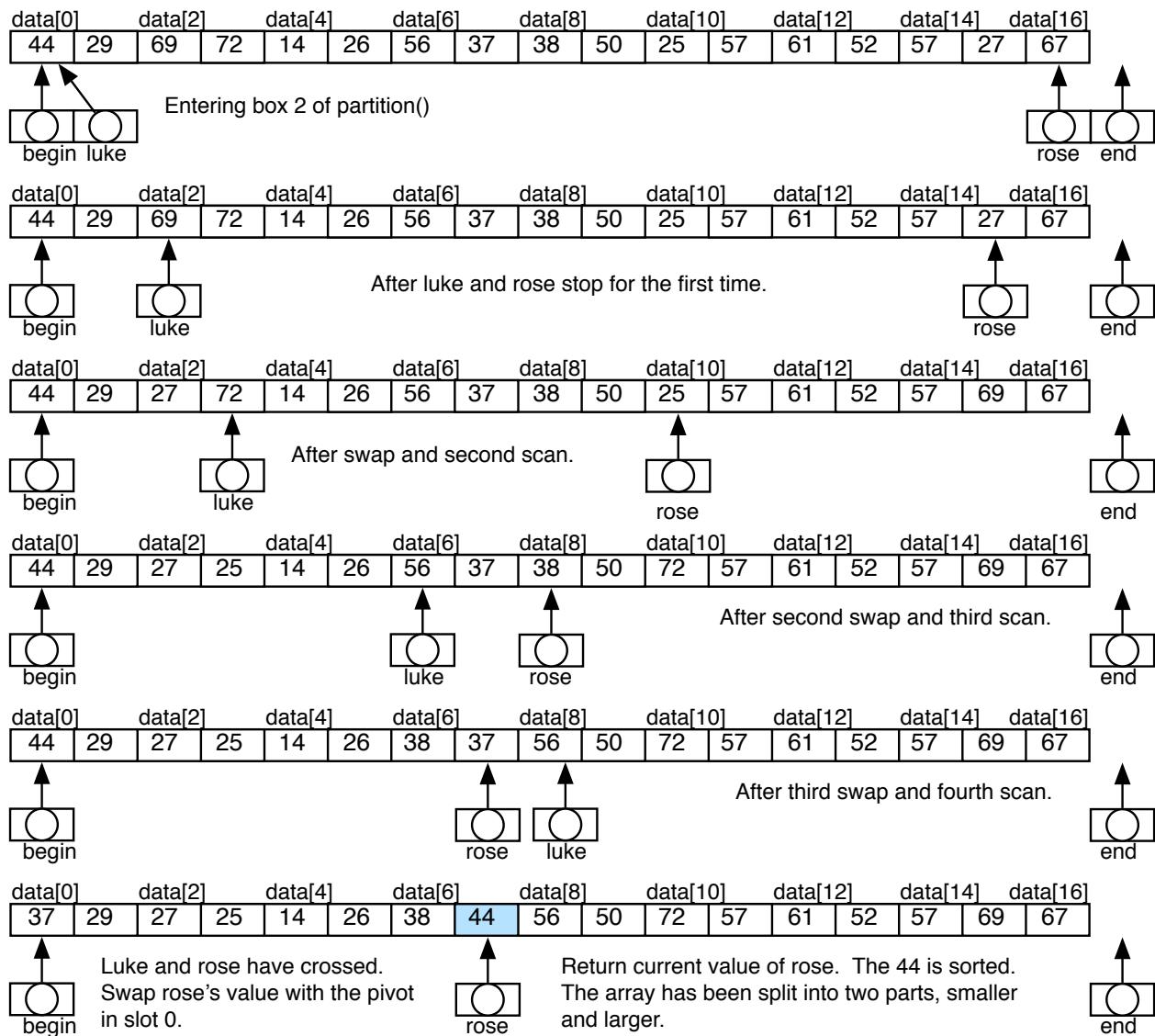


Figure 19.25. The first pass through partition.

- For modern multi-core machines with large on-board caches, the threshold seems to be 20 to 30 items.
- We can sort a small set of items (below the threshold) fastest using an insertion sort.
- By doing so, we eliminate the overhead of recursion and shorten sorting time. However, adding the insertion sort does make the code longer.

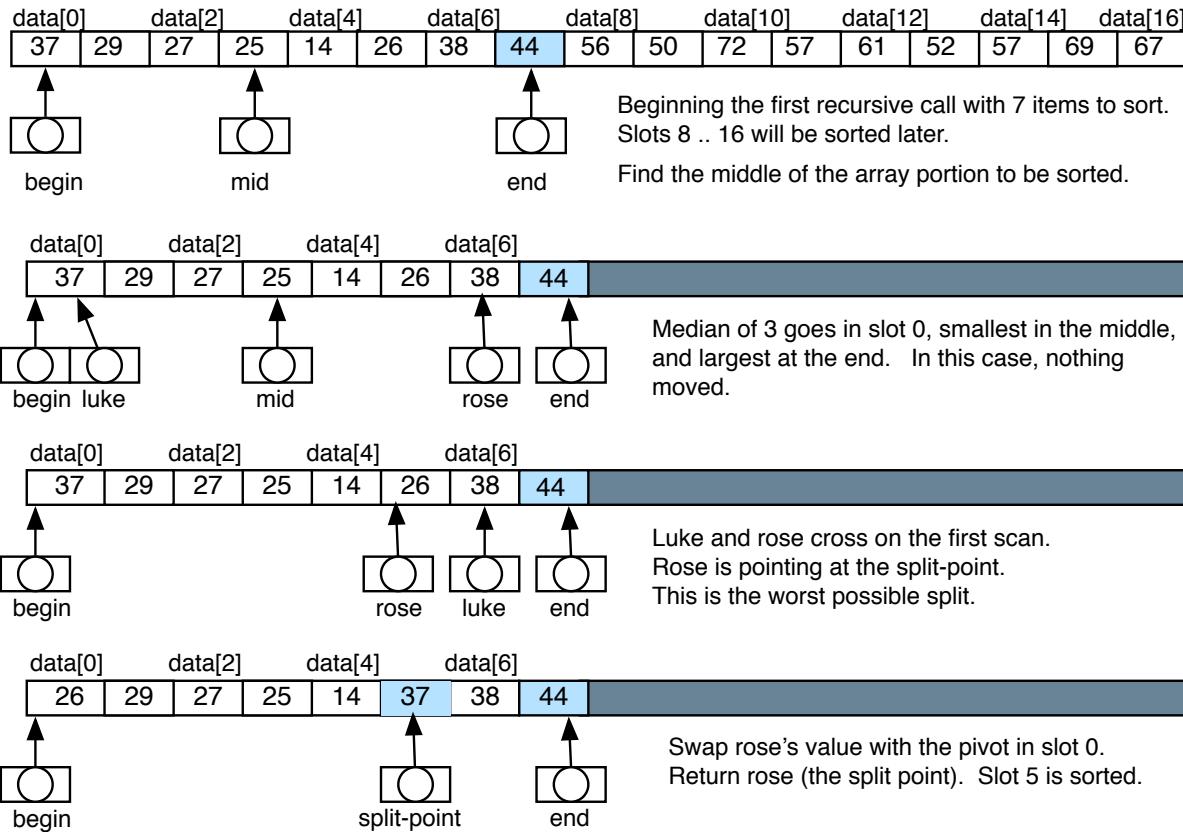
**First box, Figure 19.28: Set up three pointers for sorting.**

- **fence** is the subscript of the last unsorted data item. Items in subscripts greater than the fence are sorted. We set it to subscript  $N - 1$ , since one thing is always sorted.
- If **fence < begin**, there is nothing left to sort and we end the sort loop.
- Otherwise, we set **newcomer = \*fence**, the data at the fence.
- Now set **hole = fence** because we have copied the data and it is OK to move some other data item into that hole.
- Finally, a pointer **k** is set to the array position just to the left of the hole.
- We are now ready to begin the outer loop of the sort.

**Second box, Figure 19.28: The double-loop insertion sort.**

- In the beginning, the last item in the array is considered sorted and the second-to-last item is the first newcomer.
- To sort  $N$  items, insertion sort makes  $N - 1$  passes over the sorted part of the array. Each pass inserts one new item (the newcomer) into the sorted part.

Assume that the CUTOFF is 5.



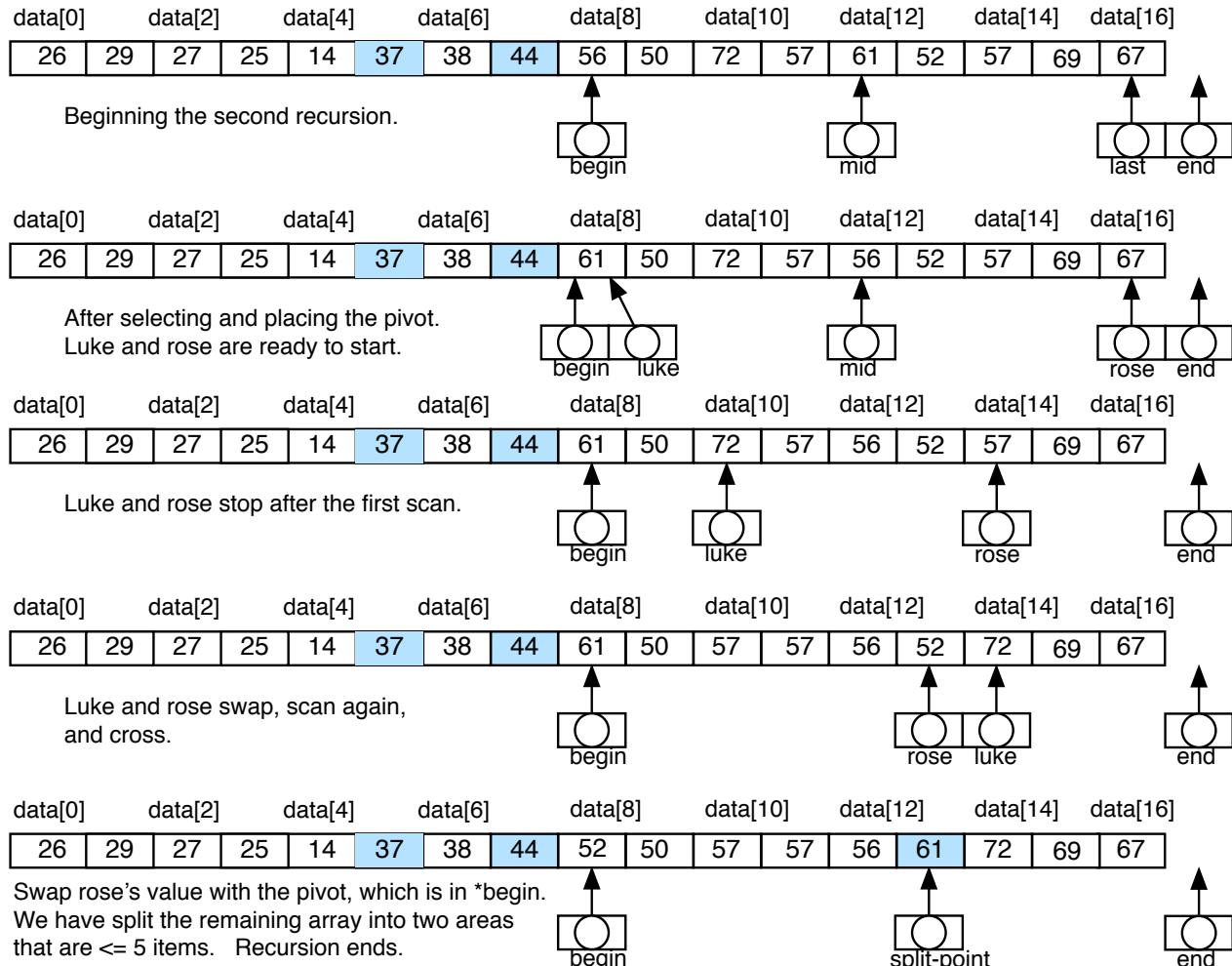
The areas to both left and right of the split point now contain five or fewer slots. Recursion ends because five is the recursion cutoff. The left end of the array is now nearly sorted and we start the recursive calls on the right end of the array.

Figure 19.26. The first recursive call.

- A pass might require only one comparison, or it might continue to the end of the array. On the average, it goes half-way to the end.
- After each trip through the inner loop, the newcomer is stored in the hole, wherever that might be.
- Note that this uses one assignment per data-move. This is much better than doing a swap, which uses three assignments.
- Control now goes back to the top of the loop, where the fence is decremented and the sorted portion of the array is lengthened by 1. Then `*fence` is copied into `newcomer` and `hole` is set =`fence`.

**Inner box, Figure 19.28:** the insertion loop.

- We insert a newcomer by moving data to the left in the array and moving the hole right, until the hole is where the newcomer belongs.
- The code is concise to the extreme: `*hole++ = *k++;`
  - This is the entire action of the loop!
  - `*hole = *k` moves the data from subscript `k` to subscript `hole`.
  - The two pointers progress in tandem. They are POST incremented, that is, they are incremented after the data is moved.



After one recursive call, there are five or fewer items on both sides of the split-point and recursion ends. The entire array is now approximately sorted. We end the recursions and use insertion sort to finish.

Figure 19.27. Finishing the recursions.

```

// Use insertion sort to finish the sorting -----
void Quick::
insort( BT* begin, BT* end ) {
    BT* hole;          // currently empty location
    BT* k;             // location currently being compared to newcomer
    BT* fence;         // last location in unsorted part
    int newcomer;       // element being inserted into sorted part

    for (fence = end - 2; fence >= begin; fence--) {
        hole = fence;
        newcomer = *hole;
        for (k = hole+1; k != end && newcomer > *k; ) {
            // Move data from slot k into the hole and ++ both indices
            *hole++ = *k++;
        }
        *hole = newcomer;
    }
}

```

Figure 19.28. Finish sorting using insertion sort.

- We really do not need two pointers here. However, using **k** eliminates the need to be constantly computing **hole-1**.
- The loop ends when the scanning pointer, **k**, reaches the end of the array or finds the insertion spot for the newcomer.
- When the insertion slot is found, the newcomer is stored in the hole.

### 19.6.1 Possible Improvements

The specific implementation just presented has been compared extensively to other variants of quicksort. It performs better in timed tests than everything else we have tried, for arrays holding tens to millions of numbers. We know of one possible significant improvement.

Execution time can be saved by using only one recursive call in **sortToCutoff**. The function would then be programmed as a loop, and instead of the second recursion, the program would assign the parameter values to local variables and go around the loop again. The loop would end when the data size was reduced to the CUTOFF.

## 19.7 What You Should Remember

### 19.7.1 Major Concepts

**Storage class.** C/C++ supports three useful storage classes: **auto**, **static**, and **extern**. Each storage class is associated with a different way of allocating and initializing storage, as well as different rules for scope and visibility.

- **auto storage.** Parameters and most local variables have storage class **auto** and are stored on the run-time stack. These objects are allocated and initialized each time the enclosing function is called and deallocated when the function completes.
- **static storage.** Global variables and **static** local variables are allocated and initialized when the program is loaded into computer memory and remain until the program terminates. The **static** variables can be

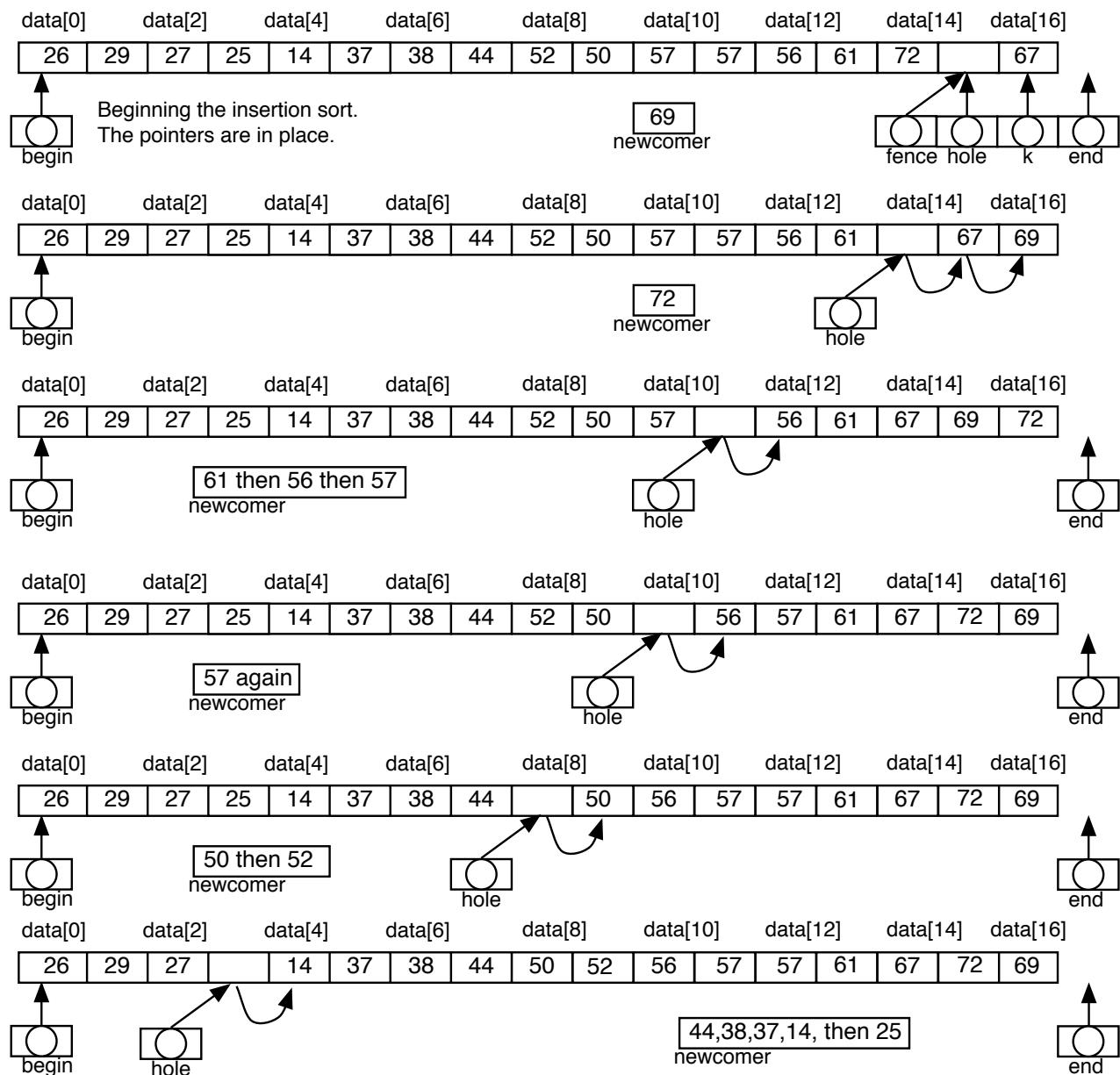


Figure 19.29. Insertion sort, first part.

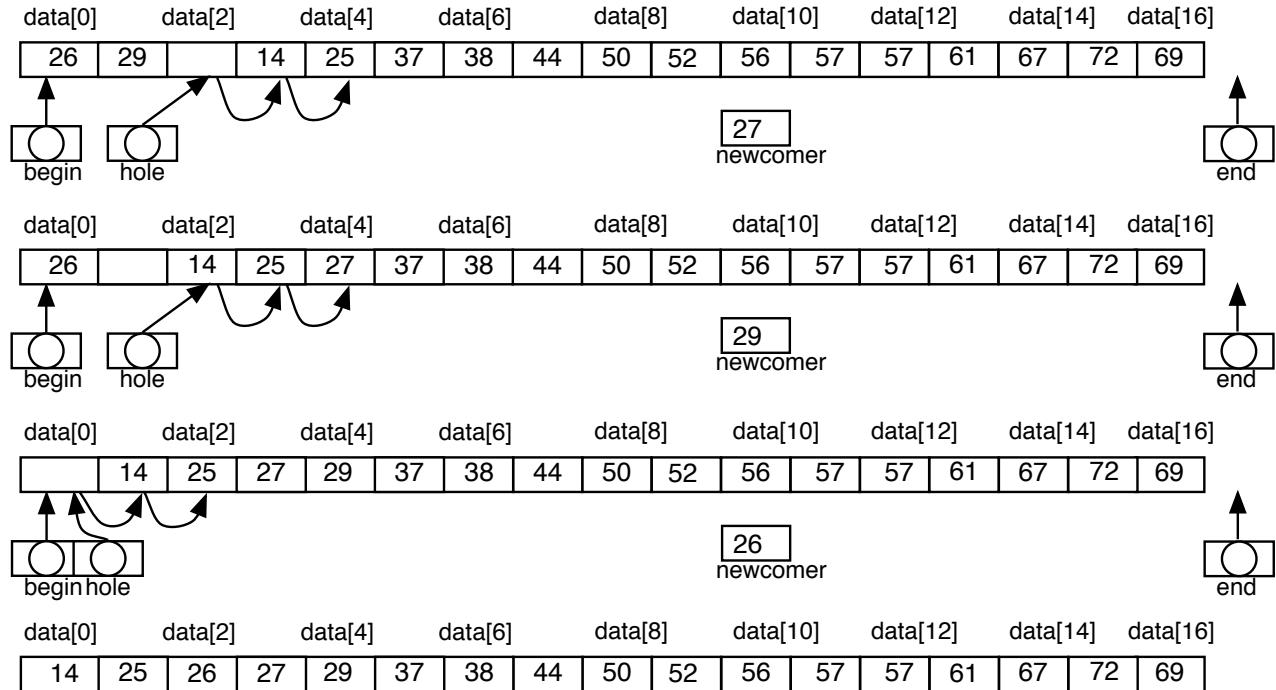


Figure 19.30. Insertion sort, second part.

used to store the state of a function between calls on that function. They also provide an efficient way to store a constant table that is local to a function.

- **extern variables.** Large applications composed of multiple code modules sometimes have `extern` variables. These are defined and initialized in one module and used in others. They remain active until the program terminates. Functions are labeled `extern` by default, but may be declared to be `static` if their scope should be restricted to one code module.

**Run-time stack.** The state of a program is kept on the run-time stack. Every active function has a stack frame that contains its parameters, local variables, and a return address. All aspects of a function call are handled through communication via the stack. Drawing stack diagrams to use while performing a program trace is one method of debugging your software.

**Recursion.** A recursive function calls itself. Using the recursive divide-and-conquer technique, we can solve a problem by dividing it into two or more simpler problems and applying the same technique again to these simpler tasks. Eventually, the subproblems become simple enough to be solved directly and the recursive descent ends. The solution to the original problem then is composed from the solutions to the simpler parts. A storage area that can grow dynamically, such as the run-time stack, is necessary to implement recursion, because multiple activation records for the same function must exist simultaneously.

**Binary search.** Binary search is the fastest way to search a sorted list using a single processor. It can be implemented using either tail recursion or iteration. The strategy is to split the list of possibilities in half, based on the relative position of the key to the middle item in the remaining search range. This is done repeatedly, until we either find the desired item or discover that it is not in the list.

**Quicksort.** This is one of the best algorithms for sorting a moderate number (up to millions) of items. The algorithm uses a recursive divide-and-conquer strategy, sorting the data by repeatedly partitioning it into a set

of small items on one end of the array and a set of large items on the other end, then recursively sorting both sets. Each partition step leaves one more item in its final sorted position between the sets of smaller and larger items.

### 19.7.2 Programming Style

- Streamline `main()`. A main program in any object-oriented language should display a greeting, create an object, call its primary function, and contain termination code. All file handling and computation should be done by class functions. This improves the modularity and overall readability of the program.
- Use private class members to model real-world objects and to store information that is calculated by one class function and used by others.
- Use local variables to store data that is used in only one function.
- Do not use global variables!
- Use `static` local variables (not globals) to implement constant tables and remember state information from one call on a function to the next. Keep the constants that belong to a function within it if possible, otherwise, define them as class members.
- Failing to check for errors at every possible stage is irresponsible. This may mean checking for invalid input parameters, checking the return values of functions, and looking for invalid operations like dividing by 0. Try to think of all the special cases your program may deal with and devise some sort of response.
- Eventually, it is important to write efficient code. However, it is more important to provide good user-information and even more important to write code that works properly. Many people spend too much time worrying about a program's efficiency and too little time worrying about getting a correct answer or appropriate error comments under all conditions.
- In general, if a problem can be solved using either iteration or tail recursion, a loop will be more efficient because it does not incur the overhead present in multiple function calls. In many situations, the recursive solution may be the most natural to write. It then can be converted to an iterative form if speed is paramount. However, for algorithms that follow a divide-and-conquer strategy, there is no easy way to avoid recursion.
- Certain functions make assumptions about the nature of the input parameters. For instance, the binary search function assumes that the data are sorted in ascending order. If you write such a function, you must write a comment explaining the precondition that you rely upon. Before your code calls such a function, you should validate such assumptions.
- Program efficiency can be improved most by looking at the amount of work done in the innermost loops. The less unnecessary work done in a loop that is executed many times, the faster the program will run. For instance, removing a test from the scanning loops of `partition()`, by using a sentinel value, makes the quicksort explained here very efficient.
- Don't use an inefficient tool if there is no need. For example, if the data array already is sorted, use a binary search rather than a linear sequential search. And, for moderate-sized data sets, use quicksort rather than insertion sort or selection sort. For small data sets, insertion sort usually is best.
- Reuse previously debugged code whenever you can.

### 19.7.3 Sticky Points and Common Errors

**Recursion.** Recursion is a technique that can be difficult to understand at first. However, once programmers understand it, they wonder what the fuss was about. To make it easier to trace a recursive program, pretend that, each time the recursive function calls itself, the new invocation has the same name but with a number appended. By having names with numbers you may be able to keep track more easily of what is going on.

**Infinite recursions.** To avoid an infinite recursive descent, a recursive algorithm must reduce the “size” of the remaining problem on every recursive step. Degenerate and basic cases must be identified and handled properly to ensure that the stopping condition eventually will become true. Just as it can be tricky to get the limits of a loop correct, a common error with recursion is to stop either one step too early or one too late.

**Storage types.** Probably the trickiest storage class to use is `static`. Mixing up `static` and `extern` can cause linker errors. Confusing `static` with `auto` may cause state information to be lost or extra copies of constants to be stored in memory.

### 19.7.4 New and Revisited Vocabulary

These are the most important terms and concepts discussed in this chapter:

storage class	run-time stack	recursive descent
<code>auto</code>	stack frame	infinite recursion
<code>extern</code>	activation record	stack overflow
<code>static</code>	parameters	list recursion
static visibility	local variables	tail recursion
static allocation area	return address	divide and conquer
state	stack diagram	binary search
initialization time	active call	split point
table of constants	iteration	quicksort
external linkage	recursion	pivot
program trace	base case(s)	partition step
	recursive step	

## 19.8 Exercises

### 19.8.1 Self-Test Exercises

1. Describe the differences between `extern` and `static` storage classes. Discuss the differences between `static` and `auto`. Try to compare `extern` and `auto` as well.
2. Sort this data set by hand into ascending order: 44, 56, 23, 14, 9, 21, 31, 8, 19, 14  
Then show how the values of `left`, `right`, and `mid` change when using the binary search algorithm to look for the value 23. Repeat this for the value 22. Use a stack diagram to show the changing parameter values for each call.
3. The following program computes the sum of the first  $N$  numbers in the array `primes`. Trace the execution of this program, showing the stack frame created for each recursive call, the values stored in the stack frame, and the value returned.

```
#include <stdio.h>
#define N 5
int sum( int ar[], int n );
int main( void )
{
    int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 };
    int answer;
    printf( "\nCalculating sum of first %i primes\n", N );
    answer = sum( primes, N );
    printf( "Sum = %i\n\n", answer );
}
// -----
int sum( int a[], int n ) {
    if (n == 1) return a[0];
    else return a[0] + sum( &a[1], n-1 );
}
```

4. The binary search function in Figure ?? has three `if` statements. These three statements could be arranged in six different orders. Say whether or not the recursion will work and terminate properly for each of the five other possibilities and explain why.
5. Take the sorted data set in Figure 19.11 and scramble it up by hand. Then show the sequence of calls on `quick()` when using the quicksort algorithm to sort the data back into ascending order. Use a stack diagram to show the changing parameter values for each call. Assume a cutoff of 3.

6. The quicksort algorithm in this chapter reorders the left half of the array, then the right half. If the order of these two recursive calls is changed, will the algorithm still work? If so, prove it. If not, explain why not.
7. The following function iteratively computes the product of the first  $n$  elements of the array  $a$ . Write a recursive version of this function. Model your solution after the `sum()` function shown in exercise 4.

```
long int product( int a[], int n )
{
    long int product = 1;
    for (int k = 0; k < n; k++) product *= a[k];
    return product;
}
```

### 19.8.2 Using Pencil and Paper

1. Consider changing the binary search algorithm in Figure ?? to work on a data set in descending order. Indicate what lines need to be changed and how.
2. Sort this data set by hand into descending order: 19, 17, 2, 43, 47, 5, 37, 23, 41, 3, 29, 31, 7, 11, 13. Then show how the value of `mid` changes when using the new binary search algorithm developed in the previous exercise. Look first for the value 3, then for the value 4. Use a stack diagram to show the changing parameter values for each call.
3. Consider changing the quicksort algorithm in this chapter so that the data set produced is in descending order. Indicate what lines need to be changed and how.
4. Scramble the sorted data set in Figure 19.11 by hand. Then show the sequence of calls on `quick()` when using the new quicksort algorithm developed in the previous exercise to sort the data into descending order. Use a stack diagram to show the changing parameter values for each call.
5. Write a recursive function that computes the value of  $n!$ . The definition of factorial was given in Chapter 7, Section 8.2.1.
6. Write a recursive function that computes the value of  $n!$ . The definition of factorial was given in Chapter 7, Section 8.2.1.
7. The following program does some string manipulation. Trace its execution, showing the stack frame created for each recursive call, the values stored in the stack frame, and the value returned.

```
#include <stdio.h>
#define N 3

char* behead( char* s, int n );

int main( void )
{
    char word[] = "distrust";
    char* answer;

    printf( "\nBeheading the victim: %s?\n", word );
    answer = behead( word, N );
    printf( "Now: %s!\n\n", answer );
}

// -----
char* behead( char* s, int n )
{
    if (n == 0) return s;
    else return behead( s+1, n-1 );
}
```

8. Write an iterative version of the `bin_search()` function in Figure ???. In many ways, the result will be similar to the `go_find()` function in Figure 12.19, which searches for the root of a function using the bisection method.

### 19.8.3 Using the Computer

1. Running sum and difference.

Write a recursive function, `sigma()`, that will return the result of an alternating sum and difference of the values in an array of `doubles`. The first, third, fifth, and so forth calls on the function should add the next array element to the total. The second, fourth, sixth, and so on calls should subtract the next array element from the total. Write a `main()` function to read in values from a user-specified data file, compute the alternating sum and difference, and print the result. Assume no more than 1,000 inputs will be processed. Hint: During debugging, print the parameter values of the recursive function and print its result just before the `return` statement.

2. Fibonacci sequence.

Write a short program that computes numbers in the Fibonacci sequence. This sequence is defined such that `fib(1) = 0` and `fib(2) = 1` and, for any other number in the sequence, `fib(n) = fib(n-1) + fib(n-2)`. Write a recursive function to compute the *n*th number in the sequence. Call this function from your main program. Use functions from the `time` library to determine how many seconds it takes to compute the 10th, 20th, and the 30th numbers in the sequence and output these times. Why does it take so long to compute the later numbers in the sequence? (See Chapter 15, computer exercise 2, for instructions on using the `time` library.)

3. Reversal.

Write a short program that inputs a word or phrase of unknown length from the user, echoes it, and then prints it backward, exactly under the original. If the phrase is a *palindrome*, it will read the same forward as backward. Use a recursive function to peel off the letters and store them in a second array in reverse order. After reversing the string, compare it to the original and display the message **Palindrome** if the string is the same forward as backward. Otherwise, display **No palindrome**. For example, *pan a nap* is a palindrome; *banana* is not. Hint: The `behead()` function (given in an earlier exercise) may help you.

4. Two piles.

Write a short program that separates negative and positive numbers, printing negative numbers on the first line and positive ones on the second line. Write a brief main program to prompt the user for a number *N* between 5 and 50. Then read *N* numbers into an array and call your recursive function (described next) to separate them. Write a recursive function named `neg_first()` to process the array. If an element is negative, the function should print it before making the next recursive call. If it is positive, the function should call itself recursively to print the other numbers and then print this number afterward.

5. Proportional search.

Modify the `bin_search()` function in Figure ?? as follows. Rather than always picking the value in the middle of the remaining array, assume that the data in the array are distributed uniformly between the smallest and largest values, and let the next index position be determined proportionally, using this formula:

$$\text{index} = \frac{\text{key} - \text{data}_{\text{first}}}{\text{data}_{\text{last}} - \text{data}_{\text{first}}} \times (\text{last} - \text{first}) + \text{first}$$

After choosing an index and testing the corresponding value, continue the search as before. Compare the performance of this algorithm with the old binary search algorithm by searching for values in the example data set used in this chapter and noting how many calls each version makes. Which is better? Why?

6. The median value.

The brute force method of finding the median value in a set is to sort the set first, then select the value in the middle. But sorting is a slow process. A more efficient algorithm, related to both quicksort and binary search, works as follows.

- (a) Let *N* be the number of values in the set, and let those values be stored in an array, **A**. If we sort the values in **A**, then by definition, the median value will be in slot *m* = *N*/2.
- (b) Perform the quicksort partitioning process on **A** once using the function in Figure 19.24. Store the return value (a pointer to the split point) in **p**.

- (c) Use pointer subtraction to compute  $s$ , the subscript corresponding to  $p$ . If  $s==m$ , we are done; we have found the median and it is in slot  $m$ . Everything to its left is smaller (or equal), everything to its right is larger, and  $m$  is in the middle of the array. If  $s$  is less than  $m$ , we know the median is *not* in the leftmost partition. Actually, it now is  $m-s$  positions from the left end of the right partition. So repeat the partition process with the right portion of the array starting at position  $s+1$  and reset  $m = m-s$ . If  $s$  is greater than  $m$ , we know the median is in the leftmost partition and still is  $m$  slots from the left end of the partition. So repeat the partition process beginning with the left end of the array and ending at position  $s-1$ .
- (d) Return the value in slot  $m$  after it has been found.

Write a program that uses this algorithm to find the median value in a user-specified file containing an unknown quantity of real numbers.

#### 7. A path through a maze.

Assume you have the following type definition:

```
typedef bool matrix[N][N];
```

where  $N$  is a #defined constant. Also assume a variable of this type is initialized with a connected trail of elements containing the value `true`, going from a source element somewhere in the middle to a border element.

- (a) Write a function, `print_trail()`, that will print out the row and column positions of the elements in this trail, from border to source, recursively. This function should have five parameters. The first is `grid`, a variable of type `matrix`. The next two are `row` and `col`; these are the subscripts of the current element of the trail. The last two are `old_row` and `old_col`, the subscripts of the preceding element. The initial call to `print_trail()` should have `row` and `col` set to the point at the beginning of the trail in the middle of `grid`, while `old_row` and `old_col` are both  $-1$ . To follow the trail, we can go up, down, right, or left. Check the four directions, looking for an element with a value `true` and making sure not to pick the one with the location `[old_row][old_col]`. Then recursively call the `print_trail()` function to follow the rest of the trail from that position on. When a border element is reached, the recursion ends. This is the base case. As the function finishes and begins returning, it should print the location of the current element. This will print the trail elements in order from the border position to the middle position.
- (b) Write a program that will read an  $N$ -by- $N$  maze matrix of ones and zeros from a user-specified file to initialize `grid`. From the next (and last) line of the file, read the row and column indexes at which the trail starts. Finally, call `print_trail()` with the starting position of the trail and have it print the trail.

# Chapter 20

## Command Line Arguments

This chapter introduces three techniques. First, we present command-line arguments, which bring directives into a main function from the operating system. Command-line arguments can be used instead of interactive or file input when executing a program from a command shell or from some IDE's.

We also introduce functions with a variable number of parameters and show how to use them to unify related function methods.

### 20.1 Command-Line Arguments

In the examples presented so far, all communication between the user and the program has been either by interactive query and response or through data stored in a file. Both are useful and powerful ways to control a program and the only ways supported by some limited systems. However, most C/C++ systems provide a way to compile a program, link in the libraries, and store an executable form for later use. Such executable files can be started from the operating system's command shell without entering the compiler's development environment. When this option is available, the operating system's command line offers one way for the user to convey control information to a program.

The **command line** is an interface between the user and the operating system. Simple information such as the number of data items to process, the name of a file, or the setting of a control switch often is passed to a program through this mechanism. A command starts with the name of the application to execute, followed by an appropriate number of additional pieces of information, each separated by spaces. Some kinds of information on the command line may be intercepted and altered or used by the system itself; for example, wild-card patterns are expanded and file redirection commands are used. The rest of the information on the line is parsed into strings and delivered to the program in the form of a **command-line argument** vector.

#### 20.1.1 The Argument Vector

To use command-line arguments, the programmer must declare parameters for `main()` to receive the following two arguments. Figure 20.1 shows an example of a command line and the argument data structure that `main()` receives because of it. This command line is for a Unix system; the `~>` is the Unix system prompt.

1. The **argument count**, an integer, customarily is named `argc`. This is the number of items, including the name of the program, that are on the command line after the operating system processes wild cards and redirection. Whitespace generally separates each item from the next. Certain grouping characters, such as quotation marks, often can be used to group separate words into one item.
2. The **argument vector**, an array of strings, customarily is named `argv`. Each element in the array is a pointer to a string containing one of the items the user wrote on the command line. (Note that this is the same data structure used for menus.) Element 0 points to the program name; the remaining elements are stored in the vector in the order they were typed.

#### 20.1.2 Decoding Arguments

The first argument (`argv[0]`) is the first thing typed on the command line: the name of the executable program. This is used for making meaningful error ~~com~~<sup>onfig</sup> messages. The other arguments can be arbitrary strings

Given the command line below, we illustrate the argument data structure delivered to `main()` by the command-line processor on a Unix system.

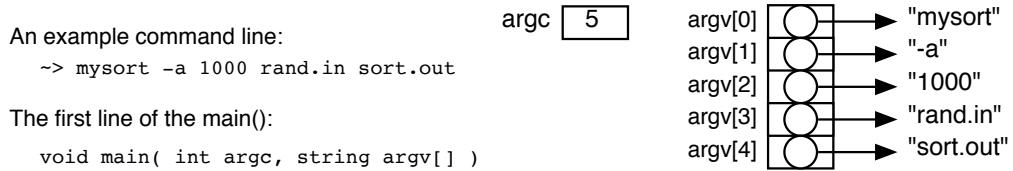


Figure 20.1. The argument vector.

or numbers; quotes are needed if a string has embedded spaces. Arguments have whatever sort of meaning the programmer devises. They have no preset meaning or order, other than that set up by the programmer. No hidden mechanism uses the command-line arguments to control a program. Customs have evolved that shed light on those issues, but those are only customs, not rules.

A program must decode and interpret the arguments itself and is well advised to check for all sorts of errors in the process. Potential errors include misspelled file names, missing or out-of-order arguments, and meaningless control switch settings. In this section, we show how a variety of command-line arguments can be decoded and used to affect the operation of a program. The program that starts in Figure 20.2 allows the user to enter an assortment of information on the command line, including a number, two file names, and (optionally) a control switch. The Sorter class is in Figures 20.3 and Figure ?? The insertion sort program here is a generalized version of the one in Figure 19.28.

#### Notes on Figure 20.2. Using command-line arguments.

This application reads a file of numbers and sorts them. The command line is used to specify the sorting order, either ascending or descending, the number of items to sort, and the names of the input and output files. These parameters must be in the order given; the sorting order is optional.

**First box, Figure 20.2:** *The #includes.* The file `tools.hpp` includes all of the C and C++ header files that a program at this level is likely to need. We `#include` it here instead of including a lot of other header files, and also to give access to the functions `banner()` and `bye()` are used in `main` to display information for the user at run time.

```
#include "tools.hpp"
#include "sorter.hpp"

// -----
int main( int argc, string argv[] ) {
    banner();
    cout <<"\nCommand Line Demo Program.\n";
    Sorter s( argc, argv );
    s.sort();
    bye();
}
```

Figure 20.2. Using command-line arguments.

**Second box, Figure 20.2 (outer): The main() function.** This function in Figure 20.2 is a typical main() program for an OO application. It prints identifying user-information at the beginning, and prints a termination comment at the end. The functions `banner()` and `bye()` are defined in `tools.cpp`.

**Second box, Figure 20.2 (inner): Entering the OO world.** Between those actions, `main()` creates one object, a Sorter. The two command-line arguments received by `main()` must be passed as arguments to the Sorter constructor. Then `main()` calls `s.sort()` to do the work, and `s.printData()` to write the sorted data to a file.

**Notes on Figure 20.3. The Sorter class.**

**First box, Figure 20.3:**

We `#include` the tools header file directly or indirectly in every other .hpp file.

**Second box, Figure 20.3: The enum declaration.**

- Enum types and enum constants are used to make code easier to read and understand. Here, we need to know whether to sort the input into ascending or descending order. We could use a boolean or integer constant, but an enum constant is better because it is not cryptic. The words “ascending” and “descending” clearly announces their meanings.
- The string array is used to create meaningful output when the enum constants are used.
- These are in the private area of the class because no other part of the program needs them.

**Third box, Figure 20.3: The data members.**

- The first four data members, `n`, `max`, `data` and `end` store the actual data and essential information about the data array: how full it is, how long it is, where it begins, and where it ends. They are initialized by the class constructor and the `getData()` function.
- The variable `order` is initialized by the constructor after decoding one of the command-line arguments. It is used by `sort()`.
- The last two data members are the input and output streams. They are opened by the constructor using the file names on the command line, and are used by `getData()` and `printData()`.

**Fourth box, Figure 20.3: The private function.**

- A function should be declared `private` when it is a helper function for some public class method.
- `getData()` is used by the destructor after the sort is finished. It is not intended for use by `main()`.
- `printData()` is used by the constructor after it finishes processing the command line. It is not intended for use by `main()`. It has a stream parameter so that it can be used for two purposes:
  - During debugging, to write the partly sorted data to the screen.
  - After sorting, to write the sorted data to the output file.

**Fifth box, Figure 20.3: The public functions.**

- The constructor and destructor for any class must be used by some function outside the class. Therefore, they are public functions.
- The constructor is defined in `sorter.cpp` because it is a long, complex method. However, the destructor is one line and is defined fully here. This is known as an `inline` function. This method writes the sorted data to the output file then frees the dynamic memory.
- `sort()` is called from `main()`. All it does is to sort the data that is already in the array.

This class is instantiated from Figure 20.2. It calls the functions in Figures ??, ??, ??, and 20.8.

```
#include "tools.hpp"

class Sorter {
private:
    enum OrderT ascend, descend ;
    string labels[2] = "ascending", "descending" ;

    int n;           // Actual number of data items read.
    int max;         // Number of data items expected.
    float* data;    // Dynamic array to store the data.
    float* end;     // Offboard end-data pointer.
    OrderT order;   // ascending or descending
    ifstream ins;
    ofstream outs;

    void getData();
    void printData( ostream& out);

public:
    Sorter( int argc, char* argv[] );
    ~Sorter(){ printData( outs ); delete[] data; }
    void sort();
};

};
```

**Figure 20.3.** The Sorter class.

#### Notes on Figure 20.4. The Sorter Constructor.

As with any constructor, the goal is to set up the data structure(s) that will be used by the rest of the Sorter class. In this case, information needed to do the setup comes from the command line and a file.

In this application, the primary inputs and outputs come from or go to files. Little or nothing needs to be displayed on the screen. However, it is disconcerting for a user to run a program and see nothing! Thus, we provide feedback at every stage of the setup process.

**First box:** The .cpp file for every class must `#include` its own .hpp file.

**Second box: Check the number of args on the command line.**

- The “-a” or “-d” switch is optional. There are also four required parameters: the name of the executable file, names for input and output files, and the number of values to read in and sort. Thus, there must be either 4 or 5 command-line arguments.
- We check whether `argc` is between 4 and 5. If not, there is an error somewhere. The customary response to this kind of error is a usage comment that lists the required and optional parameters (in square brackets).
- The function `fatal()` is in `tools.cpp`. It displays an error comment on the `cerr` stream, closes any open files, and aborts execution by calling `exit(1)`. (The ‘1’ is a code for abnormal termination.) The syntax for using `fatal()` is just like the syntax for using `printf()`. As part of the error comment you may print out one or more variables.
- This program calls `fatal()` to handle all of the errors it detects.

**Third and sixth boxes: Determine the sorting order**

- The “-a” or “-d” switch is optional. If it is present, there must be 5 arguments on the command line. If there are only 4 args, then “-a” is the default, and we set `order` = the default enum constant `ascend`.
- If there are 5 args, we must decode the optional switch. There are three possibilities that we test using C’s `strcmp()`:

- The argument matches ‘-d’, so we set `order = descend`.
- The argument matches ‘-a’, so we set `order = ascend`.
- It is something else, so we call `fatal()` and abort.
- This information is echoed back to the user in box 6.

**Fourth box, Figure 20.4: Open and verify the streams.**

- The two file names are the second and third required arguments. We open these streams.
- Whenever a stream is opened, it *must* be checked for validity. Using `streamName.is_open()` is the modern way to do this task.
- If the stream is not open, it is important to supply the name of the missing file as part of the error comment. Note how we use `fatal()` to do so.

This program calls `getData()` from Figure ??, and `fatal()` from Figure 20.8.

```
#include "sorter.hpp"

// -----
// Decode and process the command-line arguments.
Sorter::
Sorter( int argc, char* argv[] ) {
    char* last;           // To test for proper number conversion.
    int start = argc-3;   // Subscript of the first of the 3 required arguments.
    if (argc < 4 || argc > 5)
        fatal( "\nUsage: %s [-ad] num infile outfile", argv[0] );
    // ----- Pick up and check sort order (optional parameter).
    if (argc == 5) {
        if (0 == strcmp( argv[1], "-d" ) ) order = descend;
        else if (0 == strcmp( argv[1], "-a" )) order = ascend;
        else fatal( "\nSort order %s undefined.", argv[1] );
    }
    else order = ascend; // Default if switch is omitted.

    // ----- Open streams
    ins.open( argv[start + 1] );
    if (!ins.is_open()) fatal("\nCannot open %s for reading.\n", argv[start+1]);
    outs.open( argv[start + 2] );
    if (!outs.is_open()) fatal("\nCannot open %s for writing.\n", argv[start+2]);

    // ----- Convert size from string to numeric format and allocate memory.
    max = strtol( argv[start], &last, 10 );
    if (*last != '\0') fatal( "\nError: %s is not an integer.\n", argv[start] );
    if (max < 2) fatal( "\nError: %i is too few items to sort.\n", n );
    data = new float[max];
    getData();

    cout << "Data will be sorted in " <<labels[order] <<" order\n";
}
```

**Figure 20.4. The Sorter Constructor.**

***Fifth box, Figure 20.4: A number on the command line.***

- If a command-line argument is a number, it must be converted from a string of digits to a binary number before use. This is not done automatically by the system, as `cin >> n` would do for keyboard input. However, the C `stdlib` library provides the function `strtol()` for this purpose.
- `strtol()` stands for string-to-long. It converts a null-terminated array of digits to a `long int`. Its first argument is the string to be converted and the third is the conversion base (10 for decimal, 16 for hexadeciml, etc.). The second argument is the address of a `char*` that will be set by `strtol()` to the first character in the input string that is not a legal digit and was therefore not used in the conversion process. The result of the conversion is stored in `max`.
- In this context, the entire string should be a single number and the `last` pointer should be left pointing at the null terminator at the end of the string. After attempting to convert the number, which is always the third-to-last argument, we check to be sure the string was convertible and, if it was, that the result is reasonable (at least two items).
- Here are the results (with several lines omitted) of two faulty attempts at specifying the number of items:

```

~> comline 3a rand.in sort.out
Error: 3a is not an integer.
-----
~> comline a3 rand.in sort.out
Error: a3 is not an integer.

```

- If the number conversion was correct, the result is stored in `max` and used to allocate memory for the data array. Then the input function, `getData()` is called to fill the array with data.

***Notes on Figure 20.5: Reading the data.***

This function reads data items from the input file specified on the command line. Reading ends when the specified number of data items have been read, even if more data is in the file. Reading will end earlier if there is less data in the file than the expected maximum.

***First box, Figure 20.5: Mark the beginning and end.*** The input loop used here works with pointers, not subscripts. To prepare for the loop, we set two pointers: `cursor` starts at array slot 0 and traverses the array until it meets `end`, which points to the first memory location after the end of the array.

***Second box, Figure 20.5: The input loop.***

- Incrementing `cursor` (which points to one slot of an array) moves the `cursor` on to the next array slot. The loop ends when `cursor` points at the same location as `end`.
- The statement `ins >> *cursor` reads one float value from the open input file and stores it in the slot under the `cursor`.
- It is necessary to check for end-of-file *after* every read. It is not correct to check before the read. That will, in general, cause the last line to be read twice.
- It is also necessary to check for read errors after every read. There are a variety of things an application might do about a read error. In this demo program, we do the simplest thing, that is, stop reading and process whatever data we already have.
- A stream is “good” after a read if data was read, converted, and stored in memory correctly. Any kind of failure (hardware, number conversion, eof) will set a flag in the stream’s object and that will cause the stream to be not good.

***Third box Figure 20.5: How many items were read?***

- When two pointers point at slots in the same array, subtracting the leftmost from the rightmost will tell you how many slots lie between the pointers.
- Since `data` points at the slot 0 of the array and `cursor` points at the first un-filled slot, the difference `cursor - data` tells us the number of data items that were read and stored in the array. We save this number in a class member, `n` so that the `sort()` and `printData()` functions will know what part of the array to process.
- The number of data items is used to set `end` to the first array slot not occupied by data. If all expected data was read properly, this will be the first slot after the array ends. If not, it will be the first slot that does not contain valid data. This end pointer is used by the `sort()` and `printData()` functions.

---

```

void Sorter::
getData() {
    float* cursor = data;           // Set cursor to beginning of data array.
    end = data + max;              // end is an off-board end-marker.

    for( ; cursor<end; ++cursor) {
        ins >> *cursor;
        if( !ins.good() ) break; // Stop loop for error or for end of file.
    }

    n = cursor - data;            // n is the actual # of items read.
    end = data + n;               // an off-board sentinel pointer

    cout << n << " input values were read and stored.\n";
}

```

---

**Figure 20.5.** Reading the data.**Notes on Figure 20.6: Printing the sorted data.**

This function has a stream parameter so that it can send output either to the screen or to a file.

**First box:** The `cursor` is reset to the beginning of the array (slot 0). It will traverse the array until it meets `end`, which was set by the `getData()` function.

**Second box, Figure 20.6: the loop.**

- This loop is like the one in `getData()`, with one important difference: the `end` pointer might be set differently.
- Before reading data into an array, `end` must be set to mark the end of the memory locations allocated for the array. Before processing or printing, it must be set to mark the end of the data that has been stored there.

**Notes on Figure 20.7: Sort in ascending or descending order.**

An insertion sort is a nested-loop sort where the outer loop is executed a fixed number of times ( $n-1$ ). The inner loop starts with one iteration and adds one more iteration each time through the outer loop.

**First box, Figure 20.7: The pointers!**

- `fence` is the pointer that controls the outer loop.
- `hole` is the pointer that controls the inner loop and the array slot that does not contain important data.
- `newcomer` is the data that was moved out of the hole at the beginning of the inner loop.

---

```

// Print array values, one per line, to selected stream.
void Sorter::
printData( ostream& out ) {
    float* cursor = data;           // Set cursor to beginning of data array.

    for( ; cursor < end; ++cursor) out << *cursor << endl;
}

```

---

**Figure 20.6.** Printing the sorted data.

```

void Sorter::  

sort() {  

    float* hole;           // currently empty location  

    float* k;              // location currently being compared to newcomer  

    float* fence;          // last location in unsorted part  

    float newcomer;         // Data value being inserted.  

    // Insert n-1 items into the portion of array after fence, which is sorted.  

    for (fence = end - 2; fence >= data; fence--) {  

        // Pick up next item and insert into sorted portion of array.  

        hole = fence;  

        newcomer = *hole;  

        // cout <<newcomer <<" is newcomer \n";  

        k = hole + 1;  

        for (; k != end; ) {  

            // cout <<*k <<" is *k.\n";  

            if (order == ascend && newcomer <= *k )  

                break;                  // Insertion slot found...  

            else if (order == descend && newcomer >= *k)  

                break;                  // .... so leave loop.  

            else *hole++ = *k++;      // else move item back one slot.  

        }  

        *hole = newcomer;  

        // printData(cout);  

    }  

}

```

Figure 20.7. Sort in ascending or descending order.

- `k` is always positioned just to the right of `hole`. We use `k` to avoid writing `hole+1` again and again. The `+1`'s become rapidly confusing.

*Outer box, Figure 20.7: The Outer loop of Insertion sort.*

- An set of 1 thing is always sorted. So we start `fence` at the 2nd-last data value in the array. `fence` moves backward toward the head of the array. The loop ends when fence fall off the left end of the data array.
- On each iteration, the newcomer is the item under `fence`, and `hole = fence`.
- The inner loop inserts the newcomer into the part of the array to the right of the fence.
- The output statement that is commented out was used during debugging to track the progress of the process.

*First inner box, Figure 20.7: The inner loop of Insertion sort.*

- We set `k = hole+1` because we will be using both slots repeatedly as we move data from slot `k` into the hole.
- The output statement that is commented out was used during debugging to track the progress of the algorithm.
- On each iteration of the inner loop, the data in slot `k` (`*k`) moves one slot to the left in the array `*hole`, then both `k` and `hole` are incremented to move them to the next pair of slots to the right.
- This movement ends when we get to the proper insertion place for the newcomer, and break out of the loop. The “proper” slot depends on the sort order.

---

```

void fatal(const char* format, ...) {
    va_list vargs;           // optional arguments
    va_start( vargs, format );
    vfprintf( stderr, format, vargs );
    fprintf( stderr, "\nError exit; aborting.\n" );
    exit( 1 );
}

```

---

**Figure 20.8.** The `fatal()` function.

- At the end of the inner loop, the newcomer is stored in its proper position.

*Innermost box, Figure 20.7: Use the right operator for the desired sort order.*

We need a `<=` comparison to sort in ascending order and a `>=` comparison to sort in descending order.

## 20.2 Functions with Variable Numbers of Arguments

C actually provides a library, `stdarg`, to make it possible to write functions that have an indefinite number of parameters of any possible combination of types. The argument list of such a function must start with one (or more) argument that specifies how the other arguments are to be used.

For example, consider `printf()`. A call on `printf()` starts with a format and the format has one percent-sign for each expression on the rest of the list. Thus, the format tells the compiler and run-time system what to do with the remaining arguments.

The `stdarg` library allows us to write functions that work like `printf()`. We call such a function a *varargs* function. In this section, we define a varargs function, `fatal()`, to print error messages and abort execution. The new `fatal()` function accepts exactly the same set of parameters as `printf()`: a format string and a list of variables to output. It uses methods in the `stdarg` library to pass these parameters on to a library function named `vfprintf()`. When printing is done, `fatal()`, terminates execution properly.

### Notes on Figure 20.8: The `fatal()` function.

The `fatal()` function is used after an error has been detected and further progress is impossible. It takes a format argument (like `printf()`) followed by any number of data arguments. It formats and prints an error message, using the data arguments. Then it calls `exit()`, which flushes the output stream buffers, closes all open streams, and aborts cleanly.

**First box: the varargs prototype.** The parameter list of a varargs function must start with at least one ordinary (required) parameter. Following that, all of the optional parameters are replaced by three dots. This tells the compiler to pack the actual arguments into a data structure that can be passed around from function to function.

**Second box: capturing the argument list.** A function declared with `...` must declare a variable of type `va_list` and must initialize it, as shown, by calling `va_start()`. The second parameter to `va_start` is the name of the last required parameter. After this call, the variable `vargs` is initialized to point at an array of arguments suitable for processing by other varargs functions.

**Third box: using the argument list.** This line prints the error message. The function `vfprintf` expects three arguments: an open output stream, a printf-style format, and a `va_list` variable. It works exactly like `fprintf()` except that the many arguments of `fprintf()` are replaced by the single list-argument.

*Fourth box: graceful termination.*

- We wish to accomplish two things here:
  - Provide good information to the user.
  - End texecution cleanly.
- The `exit()` function flushes all output buffers and closes all open streams. Then it returns to the operating system with the error code 1, which means abnormal termination.

## 20.3 Modular Organization

A well-designed large program is built in several modules, with a `main()` program at the top level that calls functions from other modules. Each class forms a module. When the application is designed, the purpose of each class is specified, as are the ways each class can interact with the others. Each then can be stored in a separate file and developed by a different member of the development team. The modules are composed of programmer-defined classes with data and function members. Header files are used to keep the interface between the modules consistent and permit functions in one module to call functions in another. Source files contain the actual code (except for inline functions). We further explore how the techniques presented so far extend into creating larger and more protected object-oriented applications.

### 20.3.1 File Management Issues with Modular Construction

Writing, compiling, and running a simple one-module program is very much the same in any system environment. The programmer creates a source file, then either types a compile command or selects a “compile” option from a menu. If compilation is successful, an execute or run command is given either automatically, or from the command line or by selecting a menu option.

Writing and running a large program is much more complex and the process differs from system to system and from one IDE to another. The design of a large program often includes several programmer-defined modules that perform different phases of the overall task, and these may be written by different people. Each module, in turn, is a set of related definitions and functions, written in separate source code and header files, and compiled into separate object files. The object files are linked together, along with the system libraries, to create an integrated executable program. In this section, we discuss some general principles and guidelines for **modular** program **design** and explain how to build a **multimodule program** in a Unix environment.

Organizing and managing the files of a modular application and creating an executable program from them raises a group of problems related to efficiency, completeness, and consistency:

- It is undesirable to recompile an entire large application every time a change is made in one small part of it. During debugging, modules are normally changed one at a time, as errors are found. We must be able to avoid recompiling the other modules.
- We must be able to compile a module without linking it to anything, so that a debugged module can be stored much like a library.
- We must be able to link programs to object-code modules that were previously compiled, whether the module was produced locally or is part of a library package for which we do not have the source code.
- We must have access to the header files needed by each code module, whether these are our own or part of a library package.
- We must keep track of which object files and libraries are needed and make them all available to the linker at the appropriate time.
- We must avoid including the same header file twice in one module or the same source-code file twice in the linked program. The system linker will not work if it encounters a symbol that is defined twice ... even if the declarations are identical
- All modules must agree on the prototypes for shared functions, the values of common constants, the details of shared type definitions, and the names of included header files.

- We must have a way to determine whether a module is properly compiled and up to date. We must ensure that we use the most recent version of each module. A systematic way to do this is vital if the modules are being developed independently.
- If more than one programmer is working on the project, it is important to use a *version control system* to ensure that everyone works on the most recent version and that diverging changes are not introduced into the code base.
- We should have a way to test individual modules independently.

A variety of techniques have been developed to address the problems of **code management**: Effective use of subdirectories, collaboration sites, system search paths, revision control systems, makefiles, project files, header files, compiler options, and preprocessor commands, just to mention a few. These tools make it easier and faster to create a large program and much faster to debug it. The amazing large systems that we use daily could not have been developed using the tools and methods programmers had in 1970.

Different operating system environments provide alternate ways to organize files and perform the compilation tasks; the programmer must learn how each local environment works. We now discuss a few of these techniques.

**Files and folders.** Four kinds of modules might be used as part of a project:

1. *System libraries* include both the standard C or C++ libraries and compiler-specific packages for graphics, sound, and the like.
2. Local, user-defined library modules include both personal libraries and modules that are shared among members of a group or employees of a company.
3. The programmer's main module containing the function `main()`.
4. Other modules that are defined to support this particular application.

To keep all of these parts organized and avoid conflict with the parts of other programs, a subdirectory, or folder, should be established for each multimodule project. This folder will contain all of the user's code modules, header files, relevant input data and output files, and ideally, a document file that explains how to use the program. Very large projects may have a subdirectory for each module, especially if modules are being developed by different people. Depending on your system, the relevant programmer-defined library modules and header files also may go in this subdirectory. Alternatively, it may be more convenient to put local library files in a more central directory that is accessible to other projects. In this case, these files are accessed by setting up appropriate search paths. These will be in addition to the standard search paths that the compiler uses to locate system libraries and system header files.

**Header files and source code files.** The best way to work with a large application is to break it up into subsystems, then implement each subsystem as a separate code module. Each one is likely to have functions and objects for internal use only, and others that are **exported** and used by other modules. The **module interface** consists of the set of public definitions and function prototypes.

In C we organize each program module by splitting it into two sections: the internal code and the interface. The interface portion becomes a **header file** (usually with a `.h` extension). It should include the headers of the modules it depends on. The code portion becomes a **source code file** (with a `.c` extension), which must `#include` its own header file.

In C++ we organize each program module by splitting it slightly differently: the class declaration and all the `#include` commands are in the header file (with a `.hpp` or `.h` extension). This class declaration gives full definitions of very short functions (one liners). The rest of the function definitions go into the `.cpp` file, which must include its own header file.

In both C and C++, any module that uses things defined by another module must also `#include` the header file of the other module.

When we `#include` a header file in a program, the type definitions and prototypes are copied into the program. This allows the compiler to verify and set up calls on the imported functions. The actual code of those functions is *not* copied into the importing file. The code of the two files will not be connected until all modules are linked together, later.

For example, when we include `stdio.h`, we do not include the source code for the whole library (which is massively large). Rather, we include the prototypes for the library and a variety of type definitions and constants. This is the information needed to compile our calls on the library functions. Only our own code is compiled, not the library's, which saves much compilation time during a long program's development.

You should never `#include` a source code file in another module. Beginners make this error before they learn how to use an IDE and projects properly. It may work for very simple programs. However, it is not considered a good practice because it increases the possibility that functions in one code module might inadvertently interact with parts of another in unanticipated and damaging ways.

In a properly constructed modular program, the code portion of each module is compiled to produce an object code module (usually having a `.o` extension), which later is linked with the object modules of the libraries and other user modules to form a **load module**, or executable program. During the linking stage, all calls on imported functions are linked to the appropriate function in the exporting module, making a seamless and connected whole, as if the library source code actually was part of that module. The linking operation will fail if the linker cannot find a module, among the set provided, that defines each function used. It also fails if two definitions of that function are provided. For this reason, it is very important to avoid including two modules that define the same thing.

### Include-guards avoid header file duplication.

A header file may be included in several modules (normally, at least two), and it may be included by other header files. This can become a problem because a module will not compile if it includes the same header file twice. In small programs, one can often deal with this double-inclusion problem by being extremely careful. However, in a large program, the inclusion relationships can become quite complex. It is simply not worth the time and attention to be careful about what gets included where. The C/C++ preprocessor provides two tools to eliminate the multiple-inclusion problem. The easiest solution is to write

```
#pragma once
```

on the first line of the header file. This gives the compiler guidance that the contents of the file must not be included multiple times. Using `#pragma once` is convenient and easy. It works on all the systems personally known to the author, but it is *not* guaranteed by the C/C++ standard to work on all systems. Programmers using those older systems must fall back on the old way to control multiple inclusion, using `#ifndef`.

The first two commands and the last line in a header file should have the form:

```
#ifndef MYMODULE
#define MYMODULE
...
#endif
```

Following the `#pragma once`, or between the `#define` and the `#endif`, are the lines that form the interface for the module. All header files should be guarded by preprocessor commands in one way or the other. The symbol named in the `#ifndef` command is arbitrary, but it should be unique and berelated to the name of the module.

A `#ifndef` command is the beginning of a **conditional compilation** block that ends with the matching `#endif`. When the preprocessor phase of the compiler encounters the command `#ifndef MYMODULE`, it looks for `MYMODULE` in its table of defined symbols. If present, everything following the command is skipped up to the matching `#endif`, thereby ensuring that the following declarations are not included a second time. If `MYMODULE` was not previously defined, this is the first inclusion of the file. The compiler enters the block and immediately defines `MYMODULE`, preventing future double inclusion. Then it processes the other definitions and declarations in the file normally.

### 20.3.2 Building a Multimodule Program

In developing a multi-module project, it is important to have some way to specify the list of parts necessary to build that project. Also, the programmer needs a tool to help keep the parts consistent and collected in one place. In the pre-IDE days, this was done by writing a Unix *makefile*. Now, we can do it by creating a new project within the IDE and adding files to that project. Some IDEs then create a traditional *makefile* for you. The project file or *makefile* is like a top-level directory of components. It is the tool used by the IDE to guarantee that the most up-to-date version of each source file is compiled or linked each time.

**Using an IDE.** An IDE (Integrated Development Environment) for C/C++ does this job and also provides a structured text editor that “knows about” the syntax of C/C++. Other parts of the IDE supply an interface to the compiler that displays error comments, a run-time console, and a project-management system. In the Windows world, Eclipse is probably the best choice. Visual Studio is commonly used but has serious problems: it permits a variety of small errors that cause the code to not compile on a standard compiler. Simple IDE’s such as CodeBlocks are simply not adequate beyond the first programming course. For Mac users, there is XCode. Many Linux programmers use KDE or Eclipse.

**A new project.** When you ask your IDE to create a new project, the IDE will create a project folder for you, and inside it, various files and subdirectories that will be used by the IDE. Be sure this directory is created in your own home directory, not in some hidden system directory.

Many IDEs also create a code file containing a skeletal main program. This main program may be fine, as it is. However, each IDE has its own peculiarities. For example, Visual Studio puts an unnecessary line in the file that is incompatible with standard C/C++, so that line must be removed: `#include <stdafx.h>`.

If you are importing existing files from other directories, you should first copy or move those files into the project directory. Then use the IDE’s menus to “add” the files to your project. When you click on the “Build” option, everything you have “added” will be compiled and the results will be linked together to form an executable program.

After importing copies of all the code files you wish to reuse, the skeletal main program is either fleshed out to form an actual main program or replaced (using copy and paste). Write a few lines of main, then compile and test the project. Repeat until done. As you develop additional modules, continue this process: write a couple of functions, then compile and test. Avoid trying to integrate large amounts of new code at once.

### 20.3.3 Using a makefile.

A makefile lists the parts necessary to build a project and the relationships among those parts. It lets you define the compiler settings that should be used.

**The project directory.** Establish a directory for this project. Everything related to the project will go into this directory: code, data, documentation, and the results of compilation. DO NOT mix all your programs together in the same flat directory! Copy files you wish to reuse from other projects into this directory.

**Entering the code.** To use a makefile, you need a separate text editor: any one will do. Use whichever editor you like best, or use the editor that is part of your favored IDE. Open a new file and start typing. Use the same text editor to create your makefile (directions follow) and your data files. Work incrementally: code a bit, then compile and test.

**Compiling and linking.** In a typical Unix environment, the compilation and linking process is automated by a makefile. The steps involved are explicit and therefore easy to demonstrate. Analogous things happen within an IDE, under the control of options that can be set by the programmer. However, in an IDE, they are hidden and difficult to demonstrate.

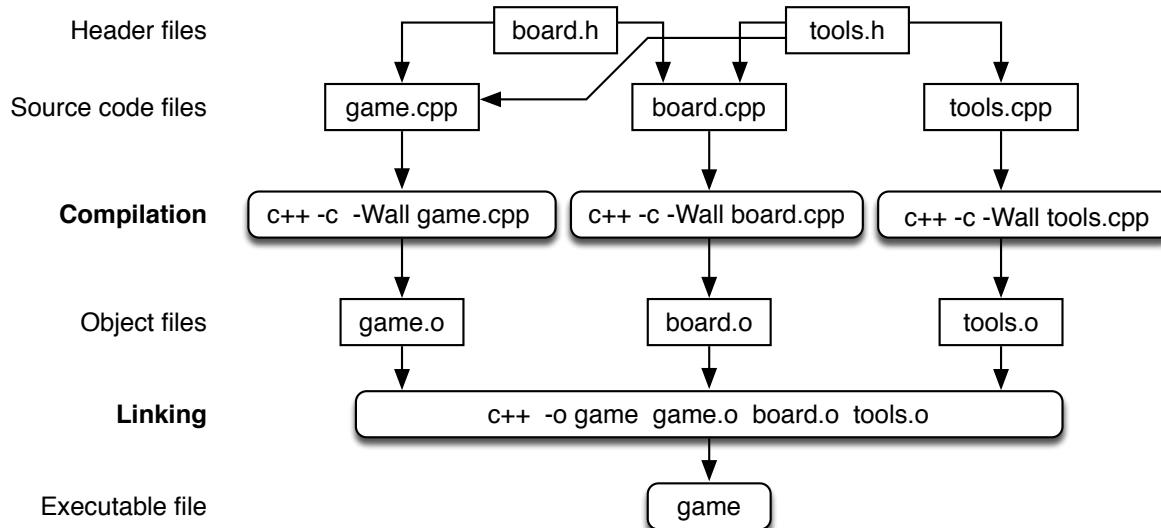
In many systems, compilers can be called directly from a command line. For instance, suppose we have a source code file, `lab1.cpp`, that we wish to **compile and link** with the standard C++ library functions. The traditional Unix command to accomplish this is

```
~> c++ -o lab1 lab1.cpp
```

The command starts with the name of the compiler (typically `c++` or `g++`). Following that is a `-o` switch and an accompanying name (`lab1`) that will be given to the executable program. The last argument is the name of the source code file. When this command is executed, the compiler is called to translate `lab1.cpp`. If there are no fatal errors, an intermediate object file, `lab1.o`, is produced. Then the linker is called to join this object file with the object files for the standard library functions. If linking also is successful, the executable file, `lab1`, is produced. To run the program, the name of the file simply is typed at the command-line prompt:

```
~> lab1
```

This diagram represents a game application with three modules (`game`, `board`, and `tools`). Rectangles represent the programmer's files, shaded oval boxes are executions of programs (the compiler, the linker, and the user's finished application). Arrows represent dependencies and the flow of information.



**Figure 20.9. Steps in building a program.**

Compilers offer many options, in addition to `-o`, which can be very useful. A complete list of **compiler options** is too numerous to give here, but we describe a few commonly used options. Here are two that you may need:

- The programs in this book were compiled under the C++ 11 standard. Some compilers default to an older standard. To use C++ 11 I must write a switch on the command line: `-std=c++11`
- Many compilers produce helpful warning comments when the `-W` switch is used to turn on “all” error checking: `-Wall`

Thus, to compile the program named `lab1.cpp`, I might write this command:

```
~> c++ -Wall -std=c++11 -o lab1 lab1.cpp
```

**Separate compilation and linking.** A single-module program generally is compiled and linked in one step. However, in a large multimodule program, each module is compiled separately, as it is developed, and linking is done as a final step. To **compile only** (without linking), the `-c` switch is used instead of `-o`. For example, suppose a game program has two user-defined modules, `game` and `board`, and also calls functions from the `tools` library. The three separate compilation commands would be

```
~> c++ -c -Wall game.c
~> c++ -c -Wall board.c
~> c++ -c -Wall tools.c
```

The upper part of the diagram in Figure 20.9 illustrates the relationships among the code files, header files, and the three compilation commands. The direct input to each step is the `.cpp` file that is listed in the `c++` command. The `.h` files are indirectly included by each `.cpp` file and so are not listed explicitly in the command.

If there are no errors in the three source code modules, the result of the compilation commands will be three object files: `game.o`, `board.o`, and `tools.o`. These three files then can be linked with functions from the standard library by using the command

```
~> c++ -o game game.o board.o tools.o
```

This step is illustrated by the bottom part of the diagram; the result is an executable program, `game`, that can be run from the command line by typing its name:

```
~> game
```

**Dependencies.** The arrows in Figure 20.9 show how the object and executable files for an application depend on various combinations of source code and header files. For example, `game.o` depends directly on `game.cpp` and through it, indirectly, on `board.h` and `tools.h`. Similarly, `tools.o` depends on `tools.cpp` and `tools.h`. The executable program, which directly depends on all three `.o` files, indirectly depends on all five source code and header files. If one of these five files is changed, every file that depends on it becomes an **obsolete file** and needs to be rebuilt. Therefore, if we change `game.cpp`, `game.o` and `game` need to be regenerated. And if we change `tools.h`, all three object files and the executable program need to be rebuilt. The `.cpp` files need not be changed because they always include the version of the `.h` files current at compilation time. Keeping all the dependent files updated is important but can be tricky, especially if an application is large and has dozens of modules. This is the kind of detailed bookkeeping that human beings find difficult but is easy for a computer. Clearly, an automated system is needed.

**A makefile automates the process.** An IDE provides automated support for making an “application” or “project.” The programmer names the project and lists the source code files and any special libraries that compose it. Once the project has been defined, the programmer typically selects a menu option labeled **MAKE** or **BUILD** to compile and link the project. Unix systems provide a “make” facility that is less user-friendly but more flexible and much more portable. The programmer creates a file of compilation and linking commands, called a **makefile**. In any system, the project file or makefile represents the application as a whole and encodes the dependency information and compilation commands shown in Figure 20.9.

A **make facility** should provide several important services:

- It should allow the programmer to list the modules and libraries that compose the application and specify how to produce and use each one.
- For each module, it should allow the programmer to list the files on which it depends. In some systems, this list is automatically derived from the source code.
- When given the command `make`, the facility should generate the executable program. Initially, this means it must compile every module and link them together.
- When a source code module is changed, the corresponding object module becomes obsolete. The next time a `make` command is given, it should be recompiled.
- When a header file is changed, the object files for all dependent modules become obsolete. The next time a `make` command is given, they should be recompiled.
- When a module is recompiled, the any executable program that uses it becomes obsolete, so the application should be relinked.

**Multiple targets.** A *target* is usually the name of a file that is generated when a program is built. Executable files and object files are examples. A target can also be the name of an action to carry out, such as ‘clean’.

A single makefile may specify several *targets*. For example, the primary target (the first one listed) may be to compile the program in the normal way, ready for deployment. A second target might compile the program in debug mode, or compile a closely related program that shares the directory. A third target (normally named “clean”) might be provided to delete all compiler output and force all modules to be recompiled on the next build.

To use a makefile to build the primary target, you simply say “make”. To build any other target defined by that makefile, write the name of the target after the “make”:

```
~> make
~> make clean
```

### 20.3.4 A Unix makefile.

A makefile contains a set of commands to compile and link an application, plus other information used to automate the process, maintain consistency, and avoid unnecessary work. A system like the Unix make facility could be implemented for any operating system. Here, though, the information we give is specific to C/C++ and Unix and systems derived from Unix.

```

# ----- Makefile for the game application

OBJS = game.o board.o tools.o
CXXFLAGS = -Wall -std=c++14 -O1
# CXX is predefined to the default C++ compiler on your machine.
# On a Mac, CXX = clang++. On Linux, CXX = g++. Both define c++ also.

# ----- Linking command

game: $(OBJS)
    $(CXX) -o game $(OBJS)

# ----- Compilation commands

game.o: game.cpp board.h tools.h
    $(CXX) -c $(CXXFLAGS) game.cpp
board.o: board.cpp board.h tools.h
    $(CXX) -c $(CXXFLAGS) board.cpp
tools.o: tools.cpp tools.h
    $(CXX) -c $(CXXFLAGS) tools.cpp

# ----- Optional cleanup command

clean:
    rm -f $(OBJS) game

```

Figure 20.10. A Unix makefile for game.

To explain the meaning and syntax for some basic commands in a makefile that accomplishes these services, we examine a makefile, shown in Figure 20.10,<sup>1</sup> for the hypothetical `game` program. We discuss the fundamental parts of a simple makefile.<sup>2</sup>

#### Notes on Figure 20.10: A Unix makefile for game.

The first line in this makefile is a comment; note that a comment is limited to one line and starts with a `#` character, not C++ comment marks. This makefile has a comment in the first box and at the beginning of each section.

##### *First box: symbol definitions.*

- A major goal of the make facility is to maintain consistency across all modules in the application. Another is to avoid omitting some essential file name or switch from one of the commands. To achieve this goal, a make facility lets us define symbolic names for phrases that must be used more than once in the set of compilation and linking commands. In this box, we define two symbols, `OBJS` and `CXXFLAGS`. The symbol `CXX` is defined by the system to be the default local C++ compiler.
- Three “flags” or “switches” are defined on this line and are recommended for use in this class:
  - `-Wall` tells the compiler to provide the highest level of warnings.
  - `-std=c++14` tells the make system to use the 2014 version of the C++ standard.
  - `-O1` tells the compiler to perform level-1 optimization. This does some flow analysis, and the comments provided may be helpful in debugging logic errors. (Note: this is a letter O, not a numeral 0.)
- These symbols are treated by the make facility in much the same way that a `const` variable is treated in C. They are defined by giving a symbol name followed by an equal sign and a defining phrase.

<sup>1</sup>All Unix-like systems provide similar make facilities; the particular one discussed here is the Gnu make program, from the Free Software Foundation.

<sup>2</sup>A full discussion of makefile capabilities is beyond the scope of this text, including archiving commands and ways to reduce the number of compilation commands that must be written.

- We define `OBJS` to stand for the list of object files that form the application. We want a symbol to stand for this list because we use the list three times in the makefile and want to be sure the three copies are identical.
- When compiling a series of modules, we should be consistent about the compiler options used. To ensure consistency, we define a symbol, `CXXFLAGS`, as the list of compiler options we want to use for the application. In this example, we use only the error warning option, `-Wall`. Note: the `XX` stands for `++`, which cannot be part of a name in this language.

***Second, third, and fourth boxes: dependencies and rules.***

These two boxes contain pairs of lines, which we will call **rules**. Each rule consists of a dependency declaration followed by a command used to compile, link, or clean up. The dependency declaration on the first line of each pair encodes the arrows in Figure 20.9. It is used by the make facility to determine whether a dependent file is current or obsolete. (A file becomes obsolete whenever a file above it in the dependency tree is changed.)

The second line of each rule is indented with a tab character. It must be a tab, not spaces. This line gives the Unix command that should be executed in order to make the target (the left side of first line) out of the code file and header files on the right.

***Second box: the linking rule.***

- The primary target should be the first target in the makefile, so its rule is the first rule in the makefile.
- The first line of this rule starts with the name that will be given to the finished application. Following that is a colon and a list of all the object files on which the application directly depends. In this example, we name the three object files by referring to the symbol `OBJS`, defined in the first box.
- The second line is a linking command. This command says that the object files listed in the first box must be linked with the standard C++ libraries to form the `game` application.
- The `$(...)` notation indicates that the symbol in the parentheses has been defined at the top of the makefile and the symbol's definition should be substituted for the `$(...)` unit. This process is almost identical to the way `#define` commands are used in C. After substitution, these lines would say:

```
game: game.o board.o tools.o
      c++ -o game game.o board.o tools.o
```

- Note: The first keystroke on the line of any compilation or linking command in a makefile must be the tab character. The make facility will not work if spaces are used to indent the `c++` command or if it is not indented.

***Third box: the compilation commands.***

- Since the application depends on three object files, there are three rules here that describe how to build them. The first line in each rule lists a set of dependencies: one source code file and all the programmer-defined header files that it includes.
- The second line in each rule is a compilation-only command. All three of these use the list of options defined for `CXXFLAGS` in the first box. After substituting the flags for the `$(...)`, the first command would say

```
game.o: game.c board.h tools.h
      c++ -c -Wall -std=c++14 -O1 game.c
```

- Using these switches, the compiler will use a low level of optimization (rather than none), give all warnings (rather than just fatal error notices), and use the C++ 2014 standard (instead of an older version of the language).

***Fourth box: the cleanup command.***

- This command automates the job of deleting all the object files and the executable file. You can have a makefile without it, but that is not a good idea.
- The phrase `rm -f` is the Unix command to remove (delete) a list of files. The `-f` flag says that the system should do it without prompting the user for verification on each file in the list. If a file is not there, the system also should not complain. We write `$(OBJS)` as an argument to delete all the object files listed in the first box. Then we add the name given to the executable file in the second box.

- The first line of the rule gives a name by which this operation can be identified. There are no prerequisites to doing the deletion command, so the dependency list is left empty. It is helpful to have a cleanup command when the programmer wishes to archive the source modules or when the file creation times indicate that the executable file is newer than any of the files on which it depends, but the programmer wishes to recompile from scratch anyway. (This can happen for a variety of reasons, such as changing the list of compiler options in `CXXFLAGS`.)
- To use the cleanup command, type

```
~> make clean
```

### Using the makefile.

For the makefile-beginner, only one makefile should be in a directory, and the files named in its dependency lists should be in the same directory. If this is true, we build the primary target by typing

```
~> make
```

The `make` program will look in the current default directory for a file named `makefile` and open it. Unless a specific rule name (such as `make clean`) has been given, it starts at the top of the makefile, interprets the definitions, and carries out the commands. It focuses on the first rule, which should be the linking rule. The name at the left side of its dependency line will be the name of the finished executable file. If that file does not exist or if it is older than one of the files on the dependency list, the linking should be redone. First, however, `make` must check whether the object files are all up-to-date.

If an object file does not exist or is obsolete, the compilation command on the second line of its rule will be used to compile the file. Compilation is skipped when the object code file is newer than the source code and header files on its dependency list. When all the required object files are up to date, the linking command in the first rule finally is executed.

## Appendix A

# The ASCII Code

The characters of the 7-bit ASCII code are listed, in order. The first two columns of each group give the code in base 10 and as a hexadecimal literal. The third column gives the printed form (if any) or a description of the character. Last is the escape code for the character, if one exists.

0	0x00	null \0	32	0x20	space	64	0x40	@	96	0x60	'
1	0x01		33	0x21	!	65	0x41	A	97	0x61	a
2	0x02		34	0x22	", \"	66	0x42	B	98	0x62	b
3	0x03		35	0x23	#	67	0x43	C	99	0x63	c
4	0x04		36	0x24	\$	68	0x44	D	100	0x64	d
5	0x05		37	0x25	%	69	0x45	E	101	0x65	e
6	0x06		38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	bell \a	39	0x27	', \'	71	0x47	G	103	0x67	g
8	0x08	backspace \b	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	tab \t	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	linefeed \n	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	vertical tab \v	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	formfeed \f	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	carriage return \r	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E		46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F		47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10		48	0x30	0	80	0x50	P	112	0x70	p
17	0x11		49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12		50	0x32	2	82	0x52	R	114	0x72	r
19	0x13		51	0x33	3	83	0x53	S	115	0x73	s
20	0x14		52	0x34	4	84	0x54	T	116	0x74	t
21	0x15		53	0x35	5	85	0x55	U	117	0x75	u
22	0x16		54	0x36	6	86	0x56	V	118	0x76	v
23	0x17		55	0x37	7	87	0x57	W	119	0x77	w
24	0x18		56	0x38	8	88	0x58	X	120	0x78	x
25	0x19		57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A		58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	escape \\	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C		60	0x3C	<	92	0x5C	\	124	0x7C	
29	0x1D		61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E		62	0x3E	>	94	0x5E	~	126	0x7E	~
31	0x1F		63	0x3F	?	95	0x5F	_	127	0x7F	delete



## Appendix B

### The Precedence of Operators in C

Arity	Operator	Meaning	Precedence	Associativity
	a[k]	Subscript	17	left to right
	fname(arg list)	Function call	17	"
	.	Struct part selection	17	"
	->	Selection using pointer	17	"
Unary	postfix ++, --	Postincrement k++, decrement k--	16	left to right
"	prefix ++, --	Preincrement ++k, decrement --k	15	right to left
"	sizeof	# of bytes in object	15	"
"	~	Bitwise complement	15	"
"	!	Logical NOT	15	"
"	+	Unary plus	15	"
"	-	Negate	15	"
"	&	Address of	15	"
"	*	Pointer dereference	15	"
"	(typename)	Type cast	14	"
Binary	*	Multiply	13	left to right
"	/	Divide	13	"
"	%	Mod	13	"
"	+	Add	12	"
"	-	Subtract	12	"
"	<<	Left shift	11	"
"	>>	Right shift	11	"
"	<	Less than	10	"
"	>	Greater than	10	"
"	<=	Less than or equal to	10	"
"	>=	Greater than or equal to	10	"
"	==	Is equal to	9	"
"	!=	Is not equal to	9	"
"	&	Bitwise AND	8	"
"	^	Bitwise exclusive OR	7	"
"		Bitwise OR	6	"
"	&&	Logical AND	5	"
"		Logical OR	4	"
Ternary	...?....:	Conditional expression	3	right to left
Binary	=	Assignment	2	"
"	+= -=	Add or subtract and store back	2	"
"	*= /= %=	Times, divide, or mod and store	2	"
"	&= ^=  =	Bitwise operator and assignment	2	"
"	<<= >>=	Shift and store back	2	"
"	,	Left-side-first sequence	1	left to right

Figure B.1. The precedence of operators in C.

# Appendix C

## Keywords

### C.1 Preprocessor Commands

The commands in the first group are presented in this text. The other commands are beyond its scope.

- Basic: `#include`, `#define`, `#ifndef`, `#endif`.
- Advanced: `#if`, `#ifdef`, `#elif`, `#else`, `defined()`, `#undef`, `#error`, `#line`, `#pragma`.
- Advanced macro operators: `#` (stringize), `##` (tokenize).

### C.2 Control Words

These words control the order of execution of program blocks.

- Functions: `main`, `return`.
- Conditionals: `if`, `else`, `switch`, `case`, `default`.
- Loops: `while`, `do`, `for`.
- Transfer of control: `break`, `continue`, `goto`.

### C.3 Types and Declarations

- Integer types: `long`, `int`, `short`, `char`, `signed`, `unsigned`.
- Real types: `double`, `float`, `long double`.
- An unknown or generic type: `void`.
- Type qualifiers: `const`, `volatile`.
- Storage class: `auto`, `static`, `extern`, `register`.
- Type operator: `sizeof`.
- To create new type names: `typedef`.
- To define new type descriptions: `struct`, `enum`, `union`.

### C.4 Additional C++ Reserved Words

The following are reserved words in C++ but not in C. C programmers should either avoid using them or be careful to use them in ways that are consistent with their meaning in C++.

- Classes: `class`, `friend`, `this`, `private`, `protected`, `public`,  
`template`.
- Functions and operators: `inline`, `virtual`, `operator`.

- Kinds of casts: `reinterpret_cast`, `static_cast`, `const_cast`, `dynamic_cast`.
- Boolean type: `bool`, `true`, `false`.
- Exceptions: `try`, `throw`, `catch`.
- Memory allocation: `new`, `delete`.
- Other: `typeid`, `namespace`, `mutable`, `asm`, `using`.

## C.5 An Alphabetical List of C and C++ Reserved Words

#	catch	goto	static
##	char	if	static_cast
#define	class	inline	struct
#elif	const	int	switch
#else	const_cast	long	template
#endif	continue	mutable	this
#error	default	namespace	throw
#if	defined()	new	true
#ifdef	delete	operator	try
#ifndef	do	private	typedef
#include	double	protected	typeid
#line	else	public	union
#pragma	enum	register	unsigned
#undef	extern	reinterpret_cast	using
asm	false	return	virtual
auto	float	short	void
bool	for	signed	volatile
break	friend	sizeof	while
case			

## Appendix D

# Advanced Aspects C Operators

This appendix describes important facts about a few C operators that were omitted in earlier chapters because they were too advanced when related material was covered.

### D.1 Assignment Combination Operators

All the assignment-combination operators have the same very low precedence and associate right to left. This means that a series of assignment operators will be parsed and executed right to left, no matter what operators are used. Figure D.1 demonstrates the syntax, precedence, and associativity of the arithmetic combinations.

**Notes on Figure D.1. Assignment combinations.**

*Box: precedence and associativity.*

- This long expression shows that all the combination operators have the same precedence and that they are parsed and executed right to left.
- The `+=` is parsed before the `*=` because it is on the right. The fact that `*` alone has higher precedence than `+` alone is not relevant to the combination operators.
- The parse tree for this expression is shown in Figure D.2. Note that assignment-combination operators have two branches connected to a variable. The right branch represents the operand value used in the mathematical operation. The left branch, with the arrowhead, reflects the changing value of the variable due to the assignment action after the calculation is complete.
- The output from this program is

```
Demonstrating Assignment Combinations
Assignment operators associate right to left.
Initial values:
    k = 10 m = 5 n = 64 t = -63
Executing t /= n -= m *= k += 7 gives the values:
    k = 17 m = 85 n = -21 t = 3
```

### D.2 More on Lazy Evaluation and Skipping

With lazy evaluation, when we skip, we skip the right operand. This isn't confusing when the right operand is only a simple variable. However, sometimes it is an expression with several operators. For example, look at the parse tree in Figure D.3. The left operand of the `||` operator is the expression `a < 10` and its right operand is `a >= 2 * b && b != 1`. The parse tree makes clear the relationship of operands to operators.

We can use parse trees to visualize the evaluation process. The stem of the tree represents the value of the entire expression. The stem of each subtree represents the value of the parts of the tree above it. To evaluate an expression, we start by writing the initial values of the variables above the expression. However, start the evaluation process *at the stem of the tree*. Starting at the top (the leaves) in C will give the wrong answer in many cases. As each operator is evaluated, write the answer on the stem under that operator. Figure D.3 illustrates the evaluation process.

The tree, as a whole, represents an assignment expression because the operator corresponding to the tree's stem is an `=`. Everything after the `=` in this assignment is a logical expression because the next operator, proceeding up the tree, is a logical operator. This is where we start considering the rules for lazy evaluation.

We exercise the arithmetic assignment-combination operators. The parse tree for the last expression is shown in Figure D.2. Note that these operators all have the same precedence and they associate right to left.

```
#include <stdio.h>
int main( void )
{
    double k = 10.0;           double m = 5.0;
    double n = 64.0;          double t = -63.0;
    puts( "\n Demonstrating Assignment Combinations" );
    puts( " Assignment operators associate right to left.\n"
          " Initial values: " );
    printf( "\t k = %.2g m = %.2g n = %.2g t = %.2g \n", k, m, n, t );
    t /= n -= m *= k += 7;
    puts( " Executing t /= n -= m *= k += 7 gives the values: " );
    printf( "\t k = %.2g m = %.2g n = %.2g t = %.2g \n\n", k, m, n, t );
}
}
```

Figure D.1. Assignment combinations.

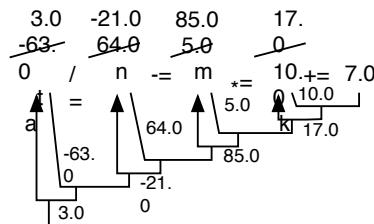
**Skip the right operand.** Evaluation of a logical expression proceeds left to right, skipping some subexpressions along the way. We evaluate the left operand of the leftmost logical operator first. Depending on the result, we evaluate or skip the right operand of that expression. In the example, we compute  $a < 10$ ; if that is **true**, we skip the rest of the expression, including the **&&** operator. This case is illustrated in the upper diagram in Figure D.3. The long double bars across the right branch of the OR operator are called *pruning marks*; they are used to “cut off” the part of the tree that is not evaluated and show, graphically, where skipping begins.

A natural comment at this point is, “But I thought that **&&** should be executed first because it has higher precedence.” Although precedence controls the construction of the parse tree, precedence simply is not considered when the tree is evaluated. Because of its higher precedence, the **&&** operator “captured” the operand  $a \geq 2 * b$ . However, logical expressions are evaluated left to right, so evaluation will start with the **||** because it is to the left of the **&&**. Only if the left operand of the **||** operation is **false**, as in the lower diagram of Figure D.3, will evaluation continue with the **&&**. In this case the left operand of the **&&** is **false**, meaning its right operand can be skipped. Graphically, evaluation starts at the stem of the logical expression and proceeds upward, doing the left side of each logical operator first and skipping the right side where possible.

**The whole right operand.** When skipping happens, *all* the work on the right subtree is skipped, no matter how complicated that portion of the expression is and no matter what operators are there. In our first example, the left operand (which we evaluated) was a simple comparison but the right operand was long and complex. As soon as we found that  $a < 10$  was **true**, we put the answer 1 on the tree stem under the **||**, skipped *all*

---

This is the parse tree and evaluation for the last expression in Figure D.1

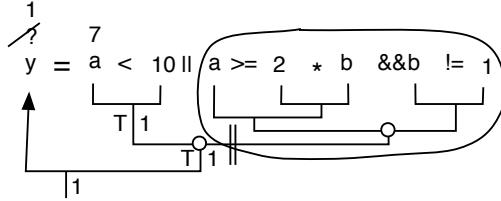



---

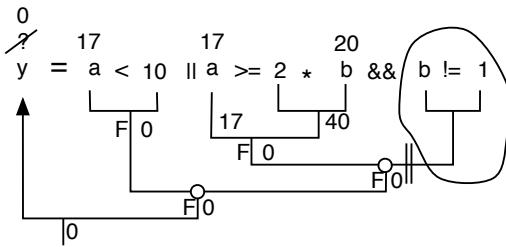
Figure D.2. A parse tree for assignment combinations.

We evaluate the expression  $y = a < 10 \text{ || } a \geq 2 * b \&& b != 1$  twice. Note the “pruning marks” on the tree and the curved lines around the parts of the expression that are skipped.

- Evaluation with the values  $a = 7, b = \text{anything}$ : The `||` causes skipping



- Evaluation with the values  $a = 17, b = 20$ : The `&&` causes skipping



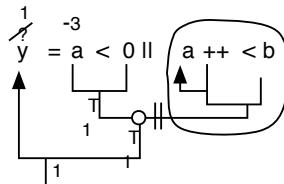
**Figure D.3. Lazy evaluation.**

the work on the right side of the operator, and stored the value 1 in  $y$ . In Figure D.4, we also skip the rest of the computation after the comparison. However, in general, we might skip only a portion of the remaining expression.

Sometimes, lazy evaluation can substantially improve the efficiency of a program. But while improving efficiency is nice, a much more important use for skipping is to avoid evaluating parts of an expression that would cause machine crashes or other kinds of trouble. For example, assume we wish to divide a number by  $x$ , compare the answer to a minimum value, and do an error procedure if the answer is less than the minimum. But it is possible for  $x$  to be 0 and that must be checked. We can avoid a division-by-0 error and do the computation and comparison in one expression by using a *guard* before the division. A guard expression consists of a test

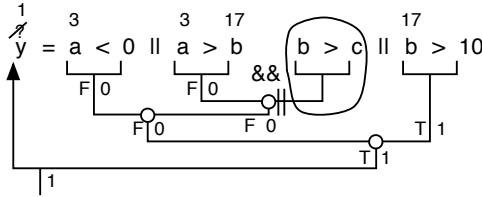
---

We evaluate the expression  $y = a < 0 \text{ || } a++ < b$  for  $a = -3$ . Note that the increment operation in the right operand of `||` does not happen because the left operand is `true`.



**Figure D.4. Skip the whole operand.**

We evaluate the expression  $y = a < 0 \text{ || } a > b \text{ && } b > c \text{ || } b > 10$  for  $a = 3$  and  $b = 17$ . Note that skipping affects only the right operand of the `&&`; the parts of the expression not on this subtree are not skipped.



**Figure D.5. And nothing but the operand.**

for the error-causing condition followed by the `&&` operator. The entire C expression would be

```
if (x != 0 && total / x < minimum) do_error();
```

Guarded expressions are useful in a wide variety of situations.

**And nothing but the right operand.** One common fallacy about C is that, once skipping starts, everything in the expression to the right is skipped. This is simply not true; the skipping involves only the right operand of the particular operator that triggered the skip. If several logical operators are in the expression, we might evaluate branches at the beginning and end but skip a part in the middle. This is illustrated by Figure D.5. In all cases, you must look at the parse tree to see what will be skipped.

### D.2.1 Evaluation Order and Side-Effect Operators

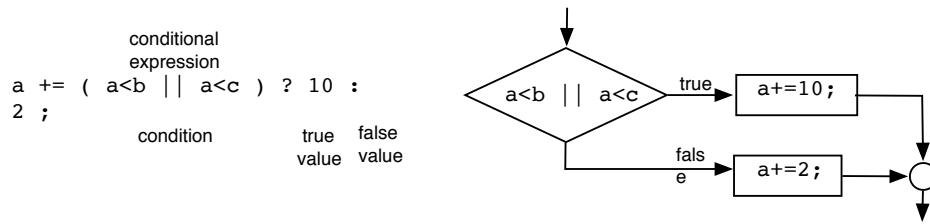
A frequent cause of confusion is the relationship between logical operators, lazy evaluation, and operators such as `++` that have side effects. When used in isolation, as at the end of Figure 4.23, the increment and decrement operators are convenient and relatively free of complication. When side-effect operators are used in long, complex expressions, they create the kind of complexity that fosters errors. If a side-effect operator is used in the middle of a logical expression, it may be executed sometimes but skipped at other times. If the operator is on the skipped subtree, as in Figure D.4, that operation is not performed and the value in memory is not changed. This may be useful in a program, but it also is complex and should be avoided by beginners. Just remember, the high precedence of the increment or decrement operator affects only the shape of the parse tree; it *does not* cause the increment operation to be evaluated before the logical operator.

A second problem with side-effect operators relates to the order in which the parts of an expression are evaluated. Recall that evaluation order has nothing to do with precedence order. We have stated that logical operators are executed left to right. This also is true of two other kinds of sequencing operators: the conditional operator `?...:` and the comma, defined in the next section. Therefore, it may be a surprise to learn that C is permitted to evaluate most other operators right-side first or left-side first, or inconsistently, whichever is convenient for the compiler. Technically, we say that the evaluation order for nonsequencing operators is *undefined*. This flexibility in evaluation order permits an optimizing compiler to produce faster code.

However, while the undefined evaluation order usually does not cause problems, it does lead directly to one important warning: If an expression contains a side-effect operator that changes the value of a variable  $V$ , *do not use  $V$  anywhere else in the expression*. The side effect could happen either before or after the value of  $V$  is used elsewhere in the expression and the outcome is unpredictable. Writing the expression in the order we want it executed won't help; the C compiler does not have to conform to our order.

## D.3 The Conditional Operator

There is only one ternary operator in C, the *conditional operator*. It has three operands and two operator symbols (`?` and `:`). The conditional operator does almost the same thing as an `if...else` with one major



**Figure D.6.** A flowchart for the conditional operator.

difference: `if` is a statement, it has no value; but `?...:` is an operator and calculates and returns a value like any other operator.

**Evaluating a Conditional Operator.** We can use either a flow diagram or a parse tree to diagram the structure and meaning of a conditional operator; each kind of diagram is helpful in some ways. A flow diagram (as in Figure D.6) depicts the order in which actions happen and shows us the similarity between a conditional operator and an `if` statement, while a parse tree (Figure D.7) shows us how the value produced by the conditional operator relates to the surrounding expression.

Making a flowchart for a conditional operator is somewhat problematical since flowcharts are for statements and a conditional operator is only part of a statement. To represent the sequence of actions as we do for the `if` statement, we have to include the rest of whatever statement contains the `?...:` in the `true` and `false` boxes. Figure D.6 shows how this can be done. The condition of the `?...:` is the operand to the left of the `?`. This condition is written in the diamond-shaped box of the flowchart. The `true` clause is written between the `?` and the `:`. It is written, with the assignment operator on the left, in the `true` box. Similarly, the `false` clause is written, with another copy of the assignment operator, in the `false` box.

Looking at the flowchart, we can see that the condition of a `?...:` always is evaluated first. Then, based on the outcome, either the `true` clause or the `false` clause is evaluated and produces a result. This result then is used in the expression that surrounds the `?...:`, in this case, a `+=` statement.

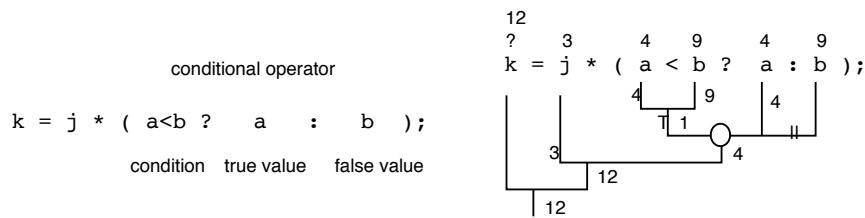
**Parsing a Conditional Operator.** Since `?...:` can be included in the middle of an expression, it is helpful to know how to draw a parse tree for it. We diagram it with three upright parts (rather than two) and a stem as shown in Figure D.7. Note that `?...:` has very low precedence (with precedence = 3, it falls just above assignment) so it usually can be used without putting parentheses around its operands. However, parentheses are often needed around the entire unit. This three-armed treelet works naturally into the surrounding expression. The main drawback of this kind of diagram is that it does not show the sequence of execution as well as a flowchart.

Parsing a nested set of conditional operators is not hard. First parse the higher-precedence parts of the expression. Then start at the right (since the conditional operator associates from right to left) and look for the pattern:

`(treelet) ? (treelet) : (treelet)`. Wherever you find three consecutive treelets separated only by a `?` and a `:`, bracket them together and draw a stem under the `?`. Even if the expression is complex and contains more than one `?`, this method always works if the expression is correct. If the expression is incorrect, we will find mismatched or misnested elements.

The sequence in which the parts of a conditional expression are evaluated or skipped is critical. We convey this sequencing in a parse tree by placing a sequence-point circle under the `?`. This indicates that the condition (the leftmost operand) must be evaluated first. The outcome of the condition selects either the `true` clause or the `false` clause and skips over the other. The skipping is conveyed by writing “prune marks” on either the middle branch or the rightmost branch of the parse tree, whichever is skipped. The expression on the remaining branch then is evaluated, and its value is written on the stem of the `?...:` bracket and propagated to the surrounding expression. Note that, even though evaluation *starts* by calculating a `true` or a `false` value, the value of the entire conditional operator, in general, will not be `true` or `false`.

The sequence point under the `?` has one other important effect. If the condition contains any postincrement operators, the increments must be done before evaluating the `true` clause or the `false` clause. Therefore, it is



**Figure D.7.** A tree for the conditional operator.

“safe” to use postincrement in a condition.

Finally, remember that evaluation order is not the same as precedence order. For example, suppose we are evaluating a conditional operator that prunes off a treelet containing some increment operators. Even though increment has much higher precedence than the conditional operator, the increment operations will not happen. This is why we must evaluate parse trees starting at the root, not the leaves. However, pruning does not change the parse tree—it merely skips part of it. We must not erase the parts that are skipped or try to get them out of the way by restructuring the whole diagram.

## D.4 The Comma Operator

The comma operator, `,`, in C is used to write two expressions in a context that normally allows for only one. To be useful, the first of these expressions must have a side effect. For example, the following loop, which sums the first `n` values in the array named `data`, uses the comma operator to initialize two variables, the loop counter and the accumulator:

```
for (sum=0, k=n-1; k>=0; --k) sum += data[k];
```

The comma operator acts much like a semicolon with two important exceptions:

1. The program units before and after a comma must be non-`void` expressions. The units before and after a semicolon can be either statements or expressions.
2. When we write a semicolon after an expression, it ends the expression and the entire unit becomes a statement. When a comma is used instead, it does not end the expression but joins it to the expression that follows to form a larger expression.
3. The value of the right operand of the comma is propagated to the enclosing expression and may be used in further computations.

## D.5 Summary

A number of nonintuitive aspects of C semantics have arisen in this appendix that are responsible for many programming errors. A programmer needs to be aware of these issues in order to use the language appropriately:

- *Use lazy evaluation.* The left operand of a logical operator always is evaluated, but the right operand is skipped whenever possible. Skipping happens when the value of the left operand is enough to determine the value of the expression.
- *Use guarded expressions.* Because of lazy evaluation, we can write compound conditionals in which the left side acts as a “guard expression” to check for and trap conditions that would cause the right side to crash. The right side is skipped if the guard expression detects a “fatal” condition.
- *Evaluation order is not the same as precedence order.* High-precedence operators are parsed first but they are not evaluated first and they may not be evaluated at all in a logical expression. Logical expressions are executed left to right with possible skipping. An operator on a part of a parse tree that is skipped will not be evaluated. Therefore, an increment operator may remain unevaluated even though it has very high precedence and the precedence of the logical operators is low.

Group	Operators	Complication
Assignment combinations	<code>+=</code> , etc.	These have low precedence and strict right-to-left parsing, no matter which combination is used.
Preincrement and predecrement	<code>++</code> , <code>--</code>	If we use a side-effect operator, we don't use the same variable again in the same expression.
Postincrement and postdecrement	<code>++</code> , <code>--</code>	Remember that the postfix operators return one value for further use in the expression and leave a different value in memory. Also, don't use the same variable again in the expression.
Logical	<code>&amp;&amp;</code> , <code>  </code> , <code>!</code>	Remember that negative integers are considered <code>true</code> values, and separate and different rules apply for precedence order and evaluation order.
	<code>&amp;&amp;</code>	Use lazy evaluation when first operand is <code>false</code> .
	<code>  </code>	Use lazy evaluation when first operand is <code>true</code> .
Conditional	<code>...?...:...</code>	The expressions both before and after the colon must produce values and those values must be the same type.
Comma	,	This is rarely used except in <code>for</code> loops.

**Figure D.8. Complications in use of side-effect operators.**

- *Evaluation order is indeterminate.* The only operators that are guaranteed to be evaluated left to right are logical-AND, logical-OR, comma, and the conditional operator. With the other binary operators, either the left side or the right side, may be evaluated first.
- *Keep side-effect operators isolated.* If you use an increment or decrement operator on a variable,  $V$ , you should not use  $V$  anywhere else in the same expression because the order of evaluation of terms in most expressions is indeterminate. If you use  $V$  again, you cannot predict whether the value of  $V$  will be changed before or after  $V$  is incremented.
- Figure D.8 summarizes the complex aspects of the C operators with side effects and sequence points.



## Appendix E

# Dynamic Allocation in C

### E.0.1 Mass Memory Allocation

When a programmer cannot predict the amount of memory that will be needed by a program, the `malloc()` function can be used at run time to allocate an array of bytes of the required size. Its prototype is

```
void* malloc( size_t sz );
```

where `size_t` is an unsigned integer type used by the local system to store the sizes of objects. Frequently, `size_t` is defined by `typedef` to be the same as `unsigned int` or `unsigned long`. The value returned by `malloc()` is a pointer to an array of bytes. For example, to allocate a single object and an array of objects of type `LumberT`, from Chapter 13, we would write:

---

These functions are defined in the C standard library whose header is `stdlib.h`.

Prototype	Action
<code>void* malloc( size_t sz );</code>	Mass memory allocation. Return a pointer to an uninitialized block of memory of the specified size, <code>sz</code> bytes.
<code>void* calloc( size_t n, size_t sz );</code>	Allocate and clear memory. Return a pointer to an array of memory locations that have been cleared to 0 bits. The array has <code>n</code> slots, each of size <code>sz</code> bytes.
<code>void free( void* pt );</code>	Recycle a memory block. Return to the operating system the block of memory that starts at the address stored in <code>pt</code> . A block should be freed after it no longer is needed by the program.
<code>void* realloc( void* pt, size_t sz );</code>	Mass memory reallocation. Given a pointer, <code>pt</code> , to a memory block that was previously allocated by <code>malloc()</code> or <code>calloc()</code> , and given a new number of bytes, <code>sz</code> , that is different from the current size of that block, resize the block to the new length. If the new block cannot start at the same location as the old one, this will involve copying the entire contents of the old block to the new one.

---

Figure E.1. Dynamic memory allocation functions.

```

lumber_t * newBoard;
lumber_t * newArray;

newBoard = malloc( sizeof(lumber_t) );
newArray = malloc( 20 * sizeof(lumber_t) );
if (newBoard == NULL || newArray == NULL) {
    fprintf( stderr, "Insufficient memory; aborting program\n" );
    exit( 1 );
}

```

Several C principles and techniques are combined in this typical code verbatim. We will now introduce and explain these principles one at a time. The simplest way to call `malloc()` is with a literal constant, thus:

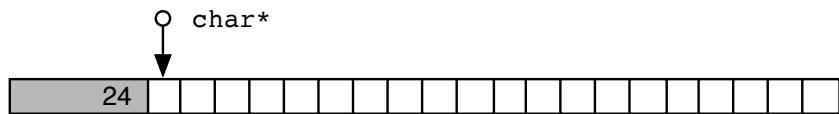
```
malloc( 20 )
```

This call allocates a memory area like the one diagrammed here:

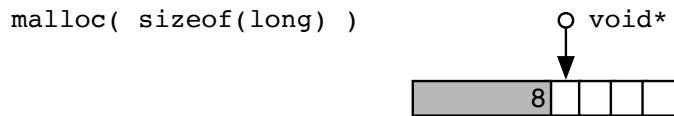


This memory is not initialized and still contains whatever data happened to be left over from a previous program. The gray area in the diagram represents additional bytes that the C system sets aside to store the total size of the allocated block (the size of a `size_t` value plus the size of the white area). The importance of these bytes becomes clear when we discuss `free()` and `realloc()`.

The type `void*` has not been discussed yet; it is a generic pointer type, which basically means “a pointer to something, but we don’t know what.” It is used because `malloc()` must be able to allocate memory for any type of object, and the function’s prototype must specify a return type compatible with all kinds of pointers. Before the `void` pointer that is returned can be used, it must be either **explicitly cast**<sup>1</sup> to a specific pointer type, as shown next, or implicitly cast by storing it in a pointer variable, as shown in Figure 16.19.



Normally, `malloc()` is used to allocate space for a single object or an array of objects of some known type. Since the number of bytes occupied by a type can vary from one implementation to another, we usually do not call `malloc()` with a literal number as an argument. Instead, we use the `sizeof` operation to supply the correct size of the desired type on the local system:

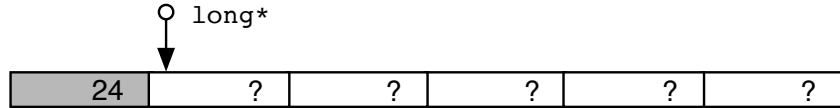


Note that the return type is still `void*`, even though we use the type name `long` in the expression. The `sizeof` expression *inside* the argument list does not affect the type of the pointer that is returned.

To allocate an array of objects, simply multiply the size of the base type by the number of array slots. The next diagram illustrates this common usage, along with a cast that converts the `void*` value to a pointer with the correct base type:

<sup>1</sup>The cast is not necessary, even to avoid warning messages, in ISO C. However, it was necessary in older versions of C and is a style that many older programmers follow.

```
(long*) malloc( 5 * sizeof(long) )
```

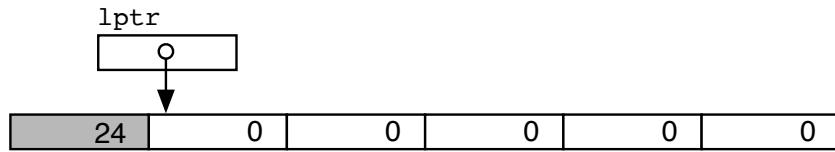


Although it is uncommon on modern systems, `malloc()` will fail if there is not enough available memory to satisfy the request. In this case, its return value is `NULL`. It is good programming practice to include a test for this condition and abort the program if it occurs, because no further meaningful processing can be done. An example of this is shown in Figure 16.19.

### E.0.2 Cleared Memory Allocation

The `calloc()` function allocates an array of memory and clears all of it to 0 bits. It has two parameters, the length of the array and the size of one array element. No cast operator is needed here because the return value is stored immediately in a pointer variable of the correct type. A typical call and its results are

```
long* lptr;
lptr = calloc( 5, sizeof(long) );
```



Even though `malloc()` and `calloc()` have different numbers of parameters, they essentially do the same thing (except for initialization) and return the same type of result. Like `malloc()`, `calloc()` returns a `void*` to the first memory address in the allocated area or returns `NULL` if not enough memory is available to allocate a block of the specified size.

### E.0.3 Freeing Dynamic Memory

In many applications, memory requirements grow and shrink repeatedly during execution. A program may request several chunks of memory to accommodate the data during one phase then, after using the memory, have no future need for it. Memory use and, sometimes, execution speed are made more efficient by recycling memory; that is, returning it to the system to be reused for some other purpose. Dynamically allocated memory can be recycled by calling `free()`, which uses the number of bytes in the gray area at the beginning of each allocated block. This function returns the block of memory to the system's memory manager, which adds it to the supply of available storage and eventually reassigns it when the program again calls `malloc()` or `calloc()`. The use of `malloc()` and `free()` are illustrated by the simulation program beginning in Figure ??.

While each program is responsible for recycling its own obsolete memory blocks, a few warnings are in order. A block should be freed only once; a second attempt to free the same block is an error. Similarly, we use `free()` only to recycle memory areas created by `malloc()` or `calloc()`. Its use with a pointer to any other memory area is an error. Another common mistake, described next, is to attempt to use a block after it has been freed. These are serious errors that cannot be detected by the compiler and may cause a variety of unpredictable results at run time.

A **dangling pointer** is one whose referent has been reclaimed by the system. Any attempt to use a dangling pointer is wrong. Typically, this happens because multiple pointers often point at the same memory block. When a block is first allocated, only one pointer points to it. However, that pointer might be copied several times as it is passed into and out of functions and stored in data structures. If one copy of the pointer is used to free the memory block, all other copies of that pointer become dangling references. A dangling reference may seem to work at first, until the block is reallocated for another purpose. After that, two parts of the program will simultaneously try to use the same storage and the contents of that location become unpredictable.

**Memory leaks.** If you do not explicitly free the dynamically allocated memory, it will be returned to the system's memory manager when the program completes. So, forgetting to perform a `free()` operation is not as damaging as freeing the same block twice.

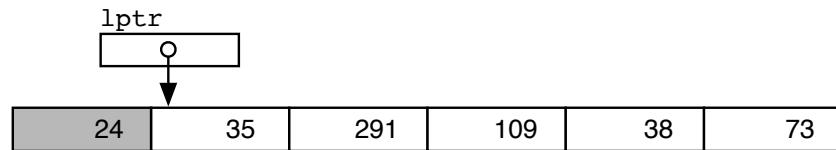
However, some programs are intended to run for hours or days at a time without being restarted. In such programs, it is much more important to keep free all dynamic memory when its useful lifetime is over. The term *memory leak* is used to describe memory that should have been recycled but was not. Memory leaks in major commercial software systems are common. The symptoms are a gradual slowdown in system performance and, eventually, a system "lock up" or crash.

Thus, it is important for programmers to learn how to free memory that is no longer needed, and it is always good programming style to do so, especially when the memory is used for a short time by only one function. Functions often are reused in a new context, they always should clean up after themselves.

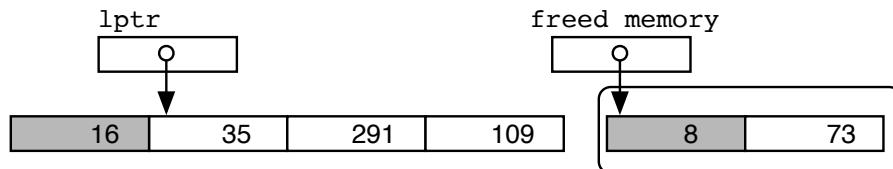
#### E.0.4 Resizing an Array

By using the `malloc()` function, we can defer the decision about the length of an array until run time. However, array space still must be allocated before the data are read and, perhaps, before we know how many data items actually exist. Using `malloc()` to create an array makes a program more flexible, but it still cannot accommodate an amount of data greater than expected. Fortunately, the C library provides an additional function to solve the problem of too little space for the data. The function that **resizes the data array** is called `realloc()`. When given a pointer to a memory block that was created by `malloc()` or `calloc()`, it will reallocate the array, making it either larger or smaller, according to the newly requested size.

Making an array smaller causes no physical or logical difficulties. The excess space is taken off the end of the original area and returned to the system for recycling. The length count that is kept in the gray area at the head of the block is adjusted appropriately. For example, assume we have the following memory block before reallocation:

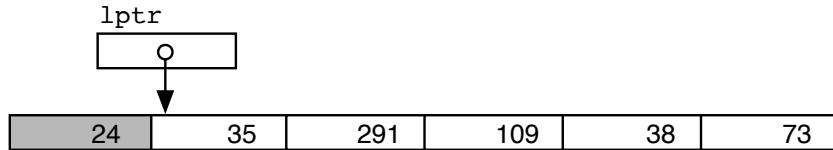


Then we give the reallocation command `lptr = realloc( lptr, 3 * sizeof(long) );`  
After reallocation, the picture becomes

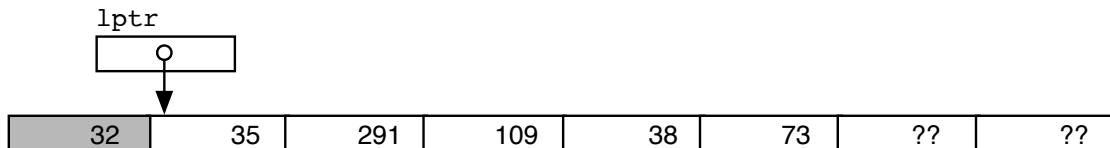


Making an array longer causes a problem because the space at the end of the array already may be in use for some other purpose. The C system keeps track of the length of every area created by `malloc()` or `calloc()` and also knows what storage is free. If there is enough empty space after the end of an array, that storage area easily and efficiently can be added onto the end. This is what `realloc()` does, if possible. When the space after the array is unavailable, `realloc()` allocates a new area that is large enough, then copies the data from the old area into the new one, and finally frees the old area.

For example, before reallocation, we have



After the reallocation command `lptr = realloc( lptr, 7 * sizeof(long) );` we have



Therefore, `realloc()` always succeeds unless the memory is full, but the reallocated array might start at a new memory address. The function returns the starting address of the current memory block, whether or not it has been changed. The program must store this address so it can access the storage.

Copying an entire array of information from one memory block to another is a costly operation and should be avoided if possible. It would not be a good idea to reallocate an array many times in a row, adding only a small number of slots each time. For this reason, successive calls on `realloc()` normally double the length of the array or at least add a substantial number of slots. An application of `realloc()` is given in the simulation program in Section 17.4.

Finally, `realloc()` is inappropriate for applications that have many pointers pointing into the dynamic array. Since the memory location of an array might change when it is reallocated, any pointers previously set to point at parts of the array are left dangling. If there are only a few such pointers, new memory addresses sometimes can be computed and the pointers reconnected properly. If not, `realloc()` should not be used.



## Appendix F

# The Standard C Environment

This appendix contains a list of standard ISO C symbols, `#include` files, and libraries. The libraries that have been used in this text are described by listing prototypes for all functions in the library. Each function is described briefly if it was used in this text or is likely to be useful to a student in the first two years of study. Alternative and traditional functions have not been listed. Readers who need more detailed information about the libraries should consult a standard reference book or the relevant **UNIX** manual page.

### F.1 Built-in Facilities

These symbols are macros that are identified at compile time and replaced by current information, as indicated. If included in a source file, the relevant information will be inserted into that file.

- `__DATE__` is the date on which the program was compiled.
- `__FILE__` is the name of the source file.
- `__LINE__` is the current line number in the source file.
- `__STDC__` is defined if the implementation conforms to the ISO C standard.
- `__TIME__` is the time at which the program was compiled and should remain constant throughout the compilation.

### F.2 Standard Files of Constants

We list the standard `#include` files that define the properties of numbers on the local system.

`limits.h`. Defines the maximum and minimum values in each of the standard integer types, as implemented on the local system. The constants defined are

- Number of bits in a character: `CHAR_BIT`.
- Type character: `CHAR_MAX`, `CHAR_MIN`.
- Signed and unsigned characters: `SCHAR_MAX`, `SCHAR_MIN`, `UCHAR_MAX`.
- Signed and unsigned short integers: `SHRT_MAX`, `SHRT_MIN`, `USHRT_MAX`.
- Signed and unsigned integers: `INT_MAX`, `INT_MIN`, `UINT_MAX`.
- Signed and unsigned long integers: `LONG_MAX`, `LONG_MIN`, `ULONG_MAX`.

---

Name	FLT_constant value	DBL_constant value
RADIX	2	
EPSILON	1.19209290E-07F	2.2204460492503131E-16
DIG	6	15
MANT_DIG	24	53
MIN	1.17549435E-38F	2.2250738585072014E-308
MIN_EXP	-125	-1021
MIN_10_EXP	-37	-307
MAX	3.40282347E+38F	1.7976931348623157E+308
MAX_EXP	128	1024
MAX_10_EXP	38	308

---

Figure F.1. Minimum values.

**float.h.** Defines the properties of each of the standard floating-point types, as implemented on the local system. The constants defined are

- The value of the radix: `FLT_RADIX`.
- Rounding mode: `FLT_ROUNDS`.
- Minimum  $x$  such that  $1.0 + x \neq x$ : `FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`.
- Decimal digits of precision: `FLT_DIG`, `DBL_DIG`, `LDBL_DIG`.
- Number of radix digits in the mantissa: `FLT_MANT_DIG`, `DBL_MANT_DIG`, `LDBL_MANT_DIG`.
- Minimum normalized positive number: `FLT_MIN`, `DBL_MIN`, `LDBL_MIN`.
- Minimum negative exponent for a normalized number: `FLT_MIN_EXP`, `DBL_MIN_EXP`, `LDBL_MIN_EXP`.
- Minimum power of 10 in the range of normalized numbers:  
`FLT_MIN_10_EXP`, `DBL_MIN_10_EXP`, `LDBL_MIN_10_EXP`.
- Maximum representable finite number: `FLT_MAX`, `DBL_MAX`, `LDBL_MAX`.
- Maximum exponent for representable finite numbers: `FLT_MAX_EXP`, `DBL_MAX_EXP`, `LDBL_MAX_EXP`.
- Maximum power of 10 for representable finite numbers:  
`FLT_MAX_10_EXP`, `DBL_MAX_10_EXP`, `LDBL_MAX_10_EXP`.

**IEEE floating-point standard.** The minimum values of the C constants that conform to the IEEE standard are listed in Figure G.1.

## F.3 The Standard Libraries and `main()`

### F.3.1 The Function `main()`

The `main()` function is special in two ways. First, every C program must have a function named `main()`, and that is where execution begins. A portion of a program may be compiled without a `main()` function, but linking will fail unless some other module does contain `main()`.

The other unique property of `main()` is that it has two official prototypes and is frequently used with others. The two standardized prototypes are:

- `int main( void );`
- `int main( int argc, char* argv[] );`

The `int` return value is intended to return a status code to the system and is needed for interprocess communication in complex applications. However, it is irrelevant for a simple program. In this case, nonstandard variants, such as `int main( void )`, can be used and will work properly. Having no return values works because most systems do not rely on a status code being returned, and simple programs have no meaningful status to report.

### F.3.2 Characters and Conversions

**Header file.** `<ctype.h>`.

**Functions.**

- `int isalnum( int ch );`  
Returns `true` if the value of `ch` is a digit (0–9) or an alphabetic character (A–Z or a–z). Returns `false` otherwise.
- `int isalpha( int ch );`  
Returns `true` if the value of `ch` is an alphabetic character (A–Z or a–z). Returns `false` otherwise.
- `int islower( int ch );`  
Returns `true` if the value of `ch` is a lower-case alphabetic character (a–z). Returns `false` otherwise.
- `int isupper( int ch );`  
Returns `true` if the value of `ch` is an upper-case alphabetic character (A–Z). Returns `false` otherwise.
- `int isdigit( int ch );`  
Returns `true` if the value of `ch` is a digit (0–9). Returns `false` otherwise.
- `int isxdigit( int ch );`  
Returns `true` if the value of `ch` is a hexadecimal digit (0–9, a–f, or A–F). Returns `false` otherwise.
- `int iscntrl( int ch );`  
Returns `true` if the value of `ch` is a control character (ASCII codes 0–31 and 127). Returns `false` otherwise.  
The complementary function for a standard ASCII implementation is `isprint()`.
- `int isprint( int ch );`  
Returns `true` if the value of `ch` is an ASCII character that is not a control character (32–126). Returns `false` otherwise.
- `int isgraph( int ch );`  
Returns `true` if the value of `ch` is a printing character other than space (33–126). Returns `false` otherwise.
- `int isspace( int ch );`  
Returns `true` if the value of `ch` is a whitespace character (horizontal tab, carriage return, newline, vertical tab, formfeed, or space). Returns `false` otherwise.
- `int ispunct( int ch );`  
Returns `true` if the value of `ch` is a printing character but not space or alphanumeric. Returns `false` otherwise.
- `int tolower( int ch );`  
If the value of `ch` is an upper-case character, returns that character converted to lower-case (a–z). Returns the letter unchanged otherwise.
- `int toupper( int ch );`  
If the value of `ch` is a lower-case character, returns that character converted to upper-case (A–Z). Returns the letter unchanged otherwise.

### F.3.3 Mathematics

**Header file.** `<math.h>`.

**Constant.** `HUGE_VAL` is the a special code that represents a value larger than the largest legal floating-point value. On some systems, if printed, it will appear as `infinity`.

**Trigonometric functions.** These functions all work in units of radians:

- `double sin( double );`  
`double cos( double );`  
`double tan( double );`

These are the mathematical functions sine, cosine, and tangent.

- `double asin( double );`  
`double acos( double );`

These functions compute the principal values of the mathematical arc sine and arc cosine functions.

- `double atan( double x );`  
`double atan2( double y, double x );`

The `atan()` function computes the principal value of the arc tangent of `x`, while `atan2()` computes the principal value of the arc tangent of `y/x`.

- `double sinh( double );`  
`double cosh( double );`  
`double tanh( double );`

These are the hyperbolic sine, cosine, and tangent functions.

### Logarithms and powers.

- `double exp( double x );`

Computes the exponential function,  $e^x$ , where  $e$  is the base of the natural logarithms.

- `double log( double x );`  
`double log10( double x );`

These are the natural logarithm and base-10 logarithm of  $x$ .

- `double pow( double x, double y );`

Computes  $x^y$ . It is an error if  $x$  is negative and  $y$  is not an exact integer or if  $x$  is 0 and  $y$  is negative or 0.

- `double sqrt( double x );`

Computes the nonnegative square root of  $x$ . It is an error if  $x$  is negative.

### Manipulating number representations.

- `double ceil( double d );`

The smallest integral value greater than or equal to  $d$ .

- `double floor( double d );`

The largest integral value less than or equal to  $d$ .

- `double fabs( double d );`

The absolute value of  $d$ . Note: `abs()` is defined in `<stdlib.h>`.

- `double fmod( double x, double y );`

The answer,  $f$ , is less than  $y$ , has the same sign as  $x$ , and  $f+y*k$  approximately equals  $x$  for some integer  $k$ . It may return 0 or be a run-time error if  $y$  is 0.

- `double frexp( double x, int* nptr );`

Splits a nonzero  $x$  into a fractional part,  $f$ , and an exponent,  $n$ , such that  $|f|$  is between 0.5 and 1.0 and  $x = f \times 2^n$ . The function's return value is  $f$ , and  $n$  is returned through the pointer parameter. If  $x$  is 0, both values will be 0.

- `double ldexp( double x, int n );`

The inverse of `frexp()`; it computes and returns the value of  $x \times 2^n$ .

- `double modf( double x, double* nptr );`

Splits a nonzero  $x$  into a fractional part,  $f$ , and an integer part,  $n$ , such that  $|f|$  is less than 1.0 and  $x = f + n$ . Both  $f$  and  $n$  have the same sign as  $x$ . The function's return value is  $f$  and  $n$  is returned through the pointer parameter.

### F.3.4 Input and Output

**Header file.** <stdio.h>.

**Predefined streams.** stdin, stdout, stderr.

**Constants.**

- EOF signifies an error or end-of-file during input.
- NULL is the zero pointer.
- FOPEN\_MAX is the number of streams that can be open simultaneously (ISO C only).
- FILENAME\_MAX is the maximum appropriate length for a file name (ISO C only).

**Types.** FILE, size\_t, fpos\_t.

**Stream functions.**

- FILE\* fopen( const char\* filename, const char\* mode );
   
int fclose( FILE\* str );
   
For opening and closing programmer-defined streams.
- int fflush( FILE\* str );
   
Sends the contents of the stream buffer to the associated device. It is defined only for output streams.
- FILE\* freopen( const char\* filename, const char\* mode,
   
FILE\* str );
   
Reopens the specified stream for the named file in the new mode.
- int feof( FILE\* str );
   
Returns true if an attempt has been made to read past the end of the stream str. Returns false otherwise.
- int ferror( FILE\* str );
   
Returns true if an error occurred while reading from or writing to the stream str.
- void clearerr( FILE\* str );
   
Resets any error or end-of-file indicators on stream str.
- int rename( const char\* oldname, const char\* newname );
   
Renames the specified disk file.
- int remove( char\* filename );
   
Deletes the named file from the disk.

**Input functions.**

- int fgetc( FILE\* str );
   
int getc( FILE\* str );
   
These functions read a single character from the stream str.
- int getchar( void );
   
Reads a single character from the stream stdin.
- int ungetc( int ch, FILE\* str );
   
Puts a single character, ch, back into the stream str.
- char\* fgets( char\* ar, int n, FILE\* str );
   
Reads up to n-1 characters from the stream str into the array ar. A newline character occurring before the nth input character terminates the operation and is stored in the array. A null character is stored at the end of the input.

- `char* gets( char* ar );`  
Reads characters from the stream `stdin` into the array `ar` until a newline character occurs, then stores a null character at the end of the input. The newline is not stored as part of the string.
- `int fscanf( FILE* str, const char* format, ... );`  
Reads input from stream `str` under the control of the format. It stores converted values in the addresses on the variable-length output list that follows the format.
- `int scanf( const char* format, ... );`  
Same as `fscanf()` to stream `stdin`.
- `int sscanf( char* s, const char* format, ... );`  
Same as `fscanf()` except that the input characters come from the string `s` instead of from a stream.
- `size_t fread( void* ar, size_t size, size_t count,
FILE* str );`  
Reads a block of data of `size` times `count` bytes from the stream `str` into array `ar`.

### Output functions.

- `int fputc( int ch, FILE* str );`  
`int putc( int ch, FILE* str );`  
These functions write `ch` to stream `str`.
- `int putchar( int ch );`  
Writes a single character, `ch`, to the stream `stdout`.
- `int fputs( const char* s, FILE* str );`  
Writes `s` to stream `str`.
- `int puts( const char* s );`  
Writes string `s` and a newline character to the stream `stdout`.
- `int fprintf( FILE* str, const char* format, ... );`  
Writes values from the variable-length output list to the stream `str` under the control of the format.
- `int printf( const char* format, ... );`  
Writes values from the variable-length output list to the stream `stdout` under the control of the format.
- `int sprintf( char* ar, const char* format, ... );`  
Writes values from the variable-length output list to the array `ar` under the control of the format.
- `size_t fwrite( const void* ar, size_t size,
size_t count, FILE* str );`  
Writes a block of data of `size` times `count` bytes from the array `ar` into the stream `str`.

**Advanced functions.** The following functions are beyond the scope of this text; their prototypes are listed without comment.

- `int setvbuf( FILE* str, char* buf, int bufmode,
size_t size );`
- `void setbuf( FILE* str, char* buf );`
- Buffer mode constants `BUFSIZ`, `_IOFBF`, `_IOLBF`, `_IONBF`
- `int fseek( FILE* str, long int offset, int from );`
- `long int ftell( FILE* str );`
- `void rewind( FILE* str );`
- Seek constants `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
- `int fgetpos( FILE* str, fpos_t* pos );`
- `int fsetpos( FILE* str, const fpos_t* pos );`

- `void perror( const char* s );`
- `int vfprintf( FILE* str, const char* format,`  
`va_list arg );`
- `int vprintf( const char* format, va_list arg );`
- `int vsprintf( char* ar, const char* format,`  
`va_list arg );`
- `FILE* tmpfile( void );`
- `char* tmpnam( char* buf );` and constants `L_tmpnam`, `TMP_MAX`

### F.3.5 Standard Library

Header file. `<stdlib.h>`.

Constants.

- `RAND_MAX`: the largest value that can be returned by `rand()`.
- `EXIT_FAILURE`: signifies unsuccessful termination when returned by `main()` or `exit()`.
- `EXIT_SUCCESS`: signifies successful termination when returned by `main()` or `exit()`.

Typedefs. `div_t` and `ldiv_t`, ISO C only, the types returned by the functions `div()` and `ldiv()`, respectively. Both are structures with two components, `quot` and `rem`, for the quotient and remainder of an integer division.

General functions.

- `int abs( int x );`  
`long labs( long x );`  
 These functions return the absolute value of `x`. Note: `fabs()`, for floating-point numbers, is defined in `<math.h>`.
- `div_t div( int n, int d );`  
`ldiv_t ldiv( long n, long d );`  
 These functions perform the integer division of `n` by `d`. The quotient and remainder are returned in a structure of type `div_t` or `ldiv_t`.
- `void srand( unsigned s );`  
`int rand( void );`  
 The function `srand()` is used to initialize the random-number generator and should be called before using `rand()`. Successive calls on `rand()` return pseudo-random numbers, evenly distributed over the range `0...RAND_MAX`.
- `void* bsearch( const void* key, const void* base,`  
`size_t count, size_t size, int (*compar)`  
`(const void* key, const void* value) );`  
 Searches the array starting at `base` for an element that matches `key`. A total of `count` elements are in the array. It uses `*compar()` to determine whether two items match. See the text for explanation.
- `int qsort( void* base, size_t count, size_t size,`  
`int (*compar)(const void* e1, const void* e2) );`  
 Quicksorts the elements of the array starting at `base` and continuing for `count` elements. It uses `*compar()` to compare the elements. See the text for explanation.

Allocation functions.

- `void* malloc( size_t size );`  
 Dynamically allocates a memory area of `size` bytes and returns the address of the beginning of this area.

- **void\* calloc( size\_t count, size\_t size );**  
Dynamically allocates a memory area of `count` times `size` bytes. It clears all the bits in this area to 0 and returns the address of the beginning of this area.
- **void free( void\* pt );**  
Returns the dynamically allocated area `*pt` to the system for future reuse.
- **void\* realloc( void\* pt, size\_t size );**  
Resizes the dynamically allocated area `*pt` to `size` bytes. If this is larger than the current size and the current allocation area cannot be extended, it allocates the entire `size` bytes elsewhere and copies the information from `*pt`.

### Control functions.

- **void exit( int status );**  
Flushes all the buffers, closes all the streams, and returns the status code to the operating system.
- **int atexit( void (\*func)( void ) );**  
ISO C only. The function `(*func)()` is called when `exit()` is called or when `main()` returns.

### String to number conversion functions.

- **double strtod( const char\* str, char\*\* p );**  
**double atof( const char\* str );**  
The function `strtod()` converts the ASCII string `str` to a number of type `double` and returns that number. It leaves `*p` pointing at the first character in `str` that was not part of the number. The function `atof()` does the same thing but does not return a pointer to the first unconverted character. The preferred function is `strtod()`; `atof()` is deprecated in the latest version of the standard.
- **long strtol( const char\* str, char\*\* p, int b );**  
The function `strtol()` converts the ASCII string `str` to a number of type `long int` expressed in base `b` and returns that number. It leaves `*p` pointing at the first character in `str` that was not part of the number. This function is preferred over both `atoi()` and `atol()`, which are deprecated in the latest version of the standard.
- **int atoi( const char\* str );**  
**long atol( const char\* str );**  
The function `atoi()` converts the ASCII string `str` to a number of type `int` expressed in base 10 and returns that number; `atol()` converts to type `long int`. The function `strtol()` is preferred over both of these, which are deprecated in the latest version of the standard.
- **unsigned long strtoul( const char\* str, char\*\* p,**  
    **int b );**  
Converts the ASCII string `str` to a number of type `long unsigned int` expressed in base `b` and returns that number. It leaves `*p` pointing at the first character in `str` that was not part of the number.

**Advanced functions.** The following functions are beyond the scope of this text; their prototypes are listed without comment.

- **void abort( void );**
- **char\* getenv( const char\* name );**
- **int system( const char\* command );**

### F.3.6 Strings

Header file. `<string.h>`.

### String manipulation.

- `char* strcat( char* dest, const char* src );`  
Appends the string `src` to the end of the string `dest`, overwriting its null terminator. We assume that `dest` has space for the combined string.
- `char* strncat(char* dest, const char* src, size_t n);`  
This function is the same as `strcat()` except that it stops after copying `n` characters, then writes a null terminator.
- `char* strcpy( char* dest, const char* src );`  
Copies the string `src` into the array `dest`. We assume that `dest` has space for the string.
- `char* strncpy(char* dest, const char* src, size_t n);`  
Copies exactly `n` characters from `src` into `dest`. If fewer than `n` characters are in `src`, null characters are appended until exactly `n` have been written.
- `int strcmp( const char* p, const char* q );`  
Compares string `p` to string `q` and returns a negative value if `p` is lexicographically less than `q`, 0 if they are equal, or a positive value if `p` is greater than `q`.
- `int strncmp(const char* p, const char* q, size_t n);`  
This function is the same as `strcmp()` but returns after comparing at most `n` characters.
- `size_t strlen( const char* s );`  
Returns the number of characters in the string `s`, excluding the null character on the end.
- `char* strchr( const char* s, int ch );`  
Searches the string `s` for the first (leftmost) occurrence of the character `ch`. Returns a pointer to that occurrence if it exists; otherwise returns `NULL`.
- `char* strrchr( const char* s, int ch );`  
Searches the string `s` for the last (rightmost) occurrence of the character `ch`. Returns a pointer to that occurrence if it exists; otherwise returns `NULL`.
- `char* strstr( const char* s, const char* sub );`  
Searches the string `s` for the first (leftmost) occurrence of the substring `sub`. Returns a pointer to the first character of that occurrence if it exists; otherwise returns `NULL`.

### Memory functions.

- `void* memchr( const void* ptr, int val, size_t len );`  
Copies `val` into `len` characters starting at address `ptr`.
- `int memcmp( const void* p, const void* q, size_t n );`  
Compares the first `n` characters starting at address `p` to the first `n` characters starting at `q`. Returns a negative value if `p` is lexicographically less than `q`, 0 if they are equal, or a positive value if `p` is greater than `q`.
- `void* memcpy(void* dest, const void* src, size_t n);`  
Copies `n` characters from `src` into `dest` and returns the address `src`. This may not work correctly for overlapping memory regions but often is faster than `memmove()`.
- `void* memmove(void* dest, const void* src, size_t n);`  
Copies `n` characters from `src` into `dest` and returns the address `src`. This works correctly for overlapping memory regions.
- `void* memset( void* ptr, int val, size_t len );`  
Copies `val` into `len` characters starting at address `ptr`.

**Advanced functions.** The following functions are beyond the scope of this text; their prototypes are listed without comment:

- `int strcoll( const char* s1, const char* s2 );`
- `size_t strcspn( const char* s, const char* set );`

- `char* strerror( int errnum );`
- `char* strpbrk( const char* s, const char* set );`
- `size_t strspn( const char* s, const char* set );`
- `char* strtok( char* s, const char* set );`
- `size_t strxfrm( char* d, const char* s, size_t len );`

### F.3.7 Time and Date

Header file. `<time.h>`.

**Constants.** `CLOCKS_PER_SEC` is the number of clock “ticks” per second of the clock used to record process time.

#### Types.

- `time_t`;  
The integer type used to represent times on the local system.
- `clock_t`;  
The arithmetic type used to represent the process time on the local system.
- `struct tm`;  
A structured representation of the time containing the following fields, all of type `int`: `tm_sec` (seconds after the minute), `tm_min` (minutes after the hour), `tm_hour` (hours since midnight, 0–23), `tm_mday` (day of the month, 1–31), `tm_mon` (month since January, 0–11), `tm_year` (years since 1900), `tm_wday` (day since Sunday, 0–6), `tm_yday` (day since January 1, 0–365), `tm_isdst` (daylight savings time flag, >0 if DST is in effect, 0 if not, <0 if unknown).

#### Functions.

- `clock_t clock()`;  
Returns an approximation to the processor time used by the current process, usually expressed in microseconds.
- `time_t time( time_t* tptr )`;  
Reads the system clock and returns the time as an integer encoding of type `time_t`. Returns the same value through the pointer parameter.
- `char* ctime( const time_t* tptr );`  
`char* asctime( const struct tm* tptr );`  
These functions return a pointer to a string containing a printable form of the date and time: "Sat Sep 14 13:12:27 1999\n". The argument to `ctime()` is a pointer to a `time_t` value such as that returned by `time()`. The argument to `asctime()` is a pointer to a structured calendar time such as that returned by `localtime()` or `gmtime()`.
- `struct tm* gmtime( const time_t* tp );`  
`struct tm* localtime( const time_t* tp );`  
These functions convert a time represented as a `time_t` value to a structured representation. The `gmtime()` returns Greenwich mean time; the `localtime()` converts to local time, taking into account the time zone and Daylight Savings Time.
- `time_t mktime( struct tm* tp );`  
Converts a time from the `struct tm` representation to the integer `time_t` representation.
- `double difftime( time_t t1, time_t t2 );`  
Returns the result of `t1-t2` in seconds as a value of type `double`.

- `size_t strftime( char* s, size_t max,  
const char* format, const struct tm* tp );`  
`size_t wcsftime( w_char* s, size_t max,  
const w_char* format, const struct tm* tp );`

The function `strftime()` formats a single date and time value specified by `tp`, storing up to `maxsize` characters into the array `s` under control of the string `format`. The function `wcsftime()` does the same thing with wide characters.

### F.3.8 Variable-Length Argument Lists

This library, known as the *vararg* facility, permits programmers to define functions with variable-length argument lists. This is an advanced feature of C and beyond the scope of this text. The list of functions is included here because this facility was used to define `say()` and `fatal()`.

**Header file.** `<stdarg.h>`.

**Type.** `va_list`.

**Functions.** `va_start`, `va_arg`, and `va_end`.

## F.4 Libraries Not Explored

Each of the remaining libraries is named and the names of functions and constants in them are listed without prototypes or explanation. This list can serve as a starting point for further exploration of C.

**Errors.** Header file: `<errno.h>`. Constants: `EDOM` and `ERANGE`.

Variable: `errno`.

**Nonlocal jumps.** Header file: `<setjmp.h>`. Type: `jmpbuf`.

Functions: `setjmp` and `longjmp`.

**Signal handling.** Header file: `<signal.h>`. Type: `sig_atomic_t`. Constants: `SIG_DFL`, `SIG_ERR`, `SIG_IGN`, `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, and `SIGTERM`. Functions: `signal`, `raise`.

**Control.** Header file: `<assert.h>`. Constant: `NDEBUG`. Function: `assert`.

**Localization.** Header file: `<locale.h>`. Constants: `LC_ALL`, `LC_TIME`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC`, `LC_COLLATE`, and `NULL`. Type: `struct lconv`. Functions: `localeconv` and `setlocale`.

**Wide-character handling.** Header file: `<wctype.h>`. Functions: `iswctype`, `towctrans`, `WEOF`, `wint_t`, `wctrans`, `wctrans_t`, `wctype`, and `wctype_t`. In addition, this library contains wide analogs of all the functions in the `ctype` library, all with a `w` as the third letter of the name: `iswupper`, `towlower`, and so on.

**Extended multibyte to wide-character conversion.** Header file: `<wchar.h>`. Functions: `btowc`, `mbrlen`, `mbtowc`, `mbstate_t`, `wcrtoutb`, `mbsinit`, `mbsrtowcs`, `wcsrtombs`, `wcstod`, `wcstol`, `wcstoul`, and `wctob`.

**Wide-string handling.** Header file: `<wchar.h>`. Functions: `wcscat`, `wcschr`, `wcscmp`, `wcsccoll`, `wcscpy`, `wcscspn`, `wcserror`, `wcslen`, `wcsncat`, `wcsncmp`, `wcsncpy`, `wcspbrk`, `wcsrchr`, `wcsspn`, `wcsstr`, `wcstok`, `wcsxfrm`, `wmemchr`, `wmemcmp`, `wmemcpy`, `wmemmove`, and `wmemset`.

**Wide-character input and output.** Header file: `<wchar.h>`. Functions: `fwprintf`, `fwscanf`, `wprintf`, `wscanf`, `swprintf`, `swscanf`, `vfwprintf`, `vwprintf`, `vwsprintf`, `fgetwc`, `fgetws`, `fputwc`, `fputws`, `getwc`, `getwchar`, `putwc`, `putwchar`, and `ungetwc`.