

BJARNE STROUSTRUP  
THE CREATOR OF C++

Using  
C++11  
and  
C++14



# PROGRAMMING

*Principles and Practice Using C++*

SECOND EDITION

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



# Programming

Second Edition

*This page intentionally left blank*



# Programming

## Principles and Practice

### Using C++

Second Edition

Bjarne Stroustrup

▼ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco  
New York • Toronto • Montreal • London • Munich • Paris • Madrid  
Capetown • Sydney • Tokyo • Singapore • Mexico City

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

A complete list of photo sources and credits appears on pages 1273–1274.

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

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the United States, please contact [international@pearsoned.com](mailto:international@pearsoned.com).

Visit us on the Web: [informati.com/aw](http://informati.com/aw)

*Library of Congress Cataloging-in-Publication Data*

Stroustrup, Bjarne, author.

Programming : principles and practice using C++ / Bjarne Stroustrup. – Second edition.

pages cm

Includes bibliographical references and index.

ISBN 978-0-321-99278-9 (pbk. : alk. paper)

1. C++ (Computer program language) I. Title.

QA76.73.C153S82 2014

005.13'3–dc23

2014004197

Copyright © 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-99278-9

ISBN-10: 0-321-99278-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.  
Second printing, January 2015

# Contents

*Preface* xxv

## Chapter 0 Notes to the Reader 1

- 0.1 The structure of this book 2
  - 0.1.1 General approach 3
  - 0.1.2 Drills, exercises, etc. 4
  - 0.1.3 What comes after this book? 5
- 0.2 A philosophy of teaching and learning 6
  - 0.2.1 The order of topics 9
  - 0.2.2 Programming and programming language 10
  - 0.2.3 Portability 11
- 0.3 Programming and computer science 12
- 0.4 Creativity and problem solving 12
- 0.5 Request for feedback 12
- 0.6 References 13
- 0.7 Biographies 13
  - Bjarne Stroustrup 14
  - Lawrence “Pete” Petersen 15

## Chapter 1 Computers, People, and Programming 17

- 1.1 Introduction 18
- 1.2 Software 19
- 1.3 People 21
- 1.4 Computer science 24
- 1.5 Computers are everywhere 25
  - 1.5.1 Screens and no screens 26
  - 1.5.2 Shipping 26
  - 1.5.3 Telecommunications 28
  - 1.5.4 Medicine 30

1.5.5	Information	31
1.5.6	A vertical view	33
1.5.7	So what?	34
1.6	Ideals for programmers	34

## Part I The Basics 41

### Chapter 2 Hello, World! 43

2.1	Programs	44
2.2	The classic first program	45
2.3	Compilation	47
2.4	Linking	51
2.5	Programming environments	52

### Chapter 3 Objects, Types, and Values 59

3.1	Input	60
3.2	Variables	62
3.3	Input and type	64
3.4	Operations and operators	66
3.5	Assignment and initialization	69
3.5.1	An example: detect repeated words	71
3.6	Composite assignment operators	73
3.6.1	An example: find repeated words	73
3.7	Names	74
3.8	Types and objects	77
3.9	Type safety	78
3.9.1	Safe conversions	79
3.9.2	Unsafe conversions	80

### Chapter 4 Computation 89

4.1	Computation	90
4.2	Objectives and tools	92
4.3	Expressions	94
4.3.1	Constant expressions	95
4.3.2	Operators	97
4.3.3	Conversions	99
4.4	Statements	100
4.4.1	Selection	102
4.4.2	Iteration	109
4.5	Functions	113
4.5.1	Why bother with functions?	115
4.5.2	Function declarations	117

4.6	<b>vector</b>	117
4.6.1	Traversing a <b>vector</b>	119
4.6.2	Growing a <b>vector</b>	119
4.6.3	A numeric example	120
4.6.4	A text example	123
4.7	Language features	125

## Chapter 5 Errors 133

5.1	Introduction	134
5.2	Sources of errors	136
5.3	Compile-time errors	136
5.3.1	Syntax errors	137
5.3.2	Type errors	138
5.3.3	Non-errors	139
5.4	Link-time errors	139
5.5	Run-time errors	140
5.5.1	The caller deals with errors	142
5.5.2	The callee deals with errors	143
5.5.3	Error reporting	145
5.6	Exceptions	146
5.6.1	Bad arguments	147
5.6.2	Range errors	148
5.6.3	Bad input	150
5.6.4	Narrowing errors	153
5.7	Logic errors	154
5.8	Estimation	157
5.9	Debugging	158
5.9.1	Practical debug advice	159
5.10	Pre- and post-conditions	163
5.10.1	Post-conditions	165
5.11	Testing	166

## Chapter 6 Writing a Program 173

6.1	A problem	174
6.2	Thinking about the problem	175
6.2.1	Stages of development	176
6.2.2	Strategy	176
6.3	Back to the calculator!	178
6.3.1	First attempt	179
6.3.2	Tokens	181
6.3.3	Implementing tokens	183
6.3.4	Using tokens	185
6.3.5	Back to the drawing board	186

6.4	Grammars	188
6.4.1	A detour: English grammar	193
6.4.2	Writing a grammar	194
6.5	Turning a grammar into code	195
6.5.1	Implementing grammar rules	196
6.5.2	Expressions	197
6.5.3	Terms	200
6.5.4	Primary expressions	202
6.6	Trying the first version	203
6.7	Trying the second version	208
6.8	Token streams	209
6.8.1	Implementing <code>Token stream</code>	211
6.8.2	Reading tokens	212
6.8.3	Reading numbers	214
6.9	Program structure	215

## Chapter 7 Completing a Program 221

7.1	Introduction	222
7.2	Input and output	222
7.3	Error handling	224
7.4	Negative numbers	229
7.5	Remainder: <code>% 230</code>	
7.6	Cleaning up the code	232
7.6.1	Symbolic constants	232
7.6.2	Use of functions	234
7.6.3	Code layout	235
7.6.4	Commenting	237
7.7	Recovering from errors	239
7.8	Variables	242
7.8.1	Variables and definitions	242
7.8.2	Introducing names	247
7.8.3	Predefined names	250
7.8.4	Are we there yet?	250

## Chapter 8 Technicalities: Functions, etc. 255

8.1	Technicalities	256
8.2	Declarations and definitions	257
8.2.1	Kinds of declarations	261
8.2.2	Variable and constant declarations	262
8.2.3	Default initialization	263

8.3	Header files	264
8.4	Scope	266
8.5	Function call and return	272
8.5.1	Declaring arguments and return type	272
8.5.2	Returning a value	274
8.5.3	Pass-by-value	275
8.5.4	Pass-by- <b>const</b> -reference	276
8.5.5	Pass-by-reference	279
8.5.6	Pass-by-value vs. pass-by-reference	281
8.5.7	Argument checking and conversion	284
8.5.8	Function call implementation	285
8.5.9	<b>constexpr</b> functions	290
8.6	Order of evaluation	291
8.6.1	Expression evaluation	292
8.6.2	Global initialization	293
8.7	Namespaces	294
8.7.1	<b>using</b> declarations and <b>using</b> directives	296
<b>Chapter 9</b>	<b>Technicalities: Classes, etc.</b>	<b>303</b>
9.1	User-defined types	304
9.2	Classes and members	305
9.3	Interface and implementation	306
9.4	Evolving a class	308
9.4.1	<b>struct</b> and functions	308
9.4.2	Member functions and constructors	310
9.4.3	Keep details private	312
9.4.4	Defining member functions	314
9.4.5	Referring to the current object	317
9.4.6	Reporting errors	317
9.5	Enumerations	318
9.5.1	“Plain” enumerations	320
9.6	Operator overloading	321
9.7	Class interfaces	323
9.7.1	Argument types	324
9.7.2	Copying	326
9.7.3	Default constructors	327
9.7.4	<b>const</b> member functions	330
9.7.5	Members and “helper functions”	332
9.8	The <b>Date</b> class	334

## Part II Input and Output 343

### Chapter 10 Input and Output Streams 345

- 10.1 Input and output 346
- 10.2 The I/O stream model 347
- 10.3 Files 349
- 10.4 Opening a file 350
- 10.5 Reading and writing a file 352
- 10.6 I/O error handling 354
- 10.7 Reading a single value 358
  - 10.7.1 Breaking the problem into manageable parts 359
  - 10.7.2 Separating dialog from function 362
- 10.8 User-defined output operators 363
- 10.9 User-defined input operators 365
- 10.10 A standard input loop 365
- 10.11 Reading a structured file 367
  - 10.11.1 In-memory representation 368
  - 10.11.2 Reading structured values 370
  - 10.11.3 Changing representations 374

### Chapter 11 Customizing Input and Output 379

- 11.1 Regularity and irregularity 380
- 11.2 Output formatting 380
  - 11.2.1 Integer output 381
  - 11.2.2 Integer input 383
  - 11.2.3 Floating-point output 384
  - 11.2.4 Precision 385
  - 11.2.5 Fields 387
- 11.3 File opening and positioning 388
  - 11.3.1 File open modes 388
  - 11.3.2 Binary files 390
  - 11.3.3 Positioning in files 393
- 11.4 String streams 394
- 11.5 Line-oriented input 395
- 11.6 Character classification 396
- 11.7 Using nonstandard separators 398
- 11.8 And there is so much more 406

### Chapter 12 A Display Model 411

- 12.1 Why graphics? 412
- 12.2 A display model 413
- 12.3 A first example 414

- 12.4 Using a GUI library 418
- 12.5 Coordinates 419
- 12.6 **Shapes** 420
- 12.7 Using **Shape** primitives 421
  - 12.7.1 Graphics headers and **main** 421
  - 12.7.2 An almost blank window 422
  - 12.7.3 **Axis** 424
  - 12.7.4 Graphing a function 426
  - 12.7.5 **Polygons** 427
  - 12.7.6 **Rectangles** 428
  - 12.7.7 Fill 431
  - 12.7.8 **Text** 431
  - 12.7.9 **Images** 433
  - 12.7.10 And much more 434
- 12.8 Getting this to run 435
  - 12.8.1 Source files 437

## Chapter 13 **Graphics Classes** 441

- 13.1 Overview of graphics classes 442
- 13.2 **Point** and **Line** 444
- 13.3 **Lines** 447
- 13.4 **Color** 450
- 13.5 **Line\_style** 452
- 13.6 **Open\_polyline** 455
- 13.7 **Closed\_polyline** 456
- 13.8 **Polygon** 458
- 13.9 **Rectangle** 460
- 13.10 Managing unnamed objects 465
- 13.11 **Text** 467
- 13.12 **Circle** 470
- 13.13 **Ellipse** 472
- 13.14 **Marked\_polyline** 474
- 13.15 **Marks** 476
- 13.16 **Mark** 478
- 13.17 **Images** 479

## Chapter 14 **Graphics Class Design** 487

- 14.1 Design principles 488
  - 14.1.1 Types 488
  - 14.1.2 Operations 490
  - 14.1.3 Naming 491
  - 14.1.4 Mutability 492

- 14.2 **Shape** 493
  - 14.2.1 An abstract class 495
  - 14.2.2 Access control 496
  - 14.2.3 Drawing shapes 500
  - 14.2.4 Copying and mutability 503
- 14.3 Base and derived classes 504
  - 14.3.1 Object layout 506
  - 14.3.2 Deriving classes and defining virtual functions 507
  - 14.3.3 Overriding 508
  - 14.3.4 Access 511
  - 14.3.5 Pure virtual functions 512
- 14.4 Benefits of object-oriented programming 513

## Chapter 15 Graphing Functions and Data 519

- 15.1 Introduction 520
- 15.2 Graphing simple functions 520
- 15.3 **Function** 524
  - 15.3.1 Default Arguments 525
  - 15.3.2 More examples 527
  - 15.3.3 Lambda expressions 528
- 15.4 **Axis** 529
- 15.5 Approximation 532
- 15.6 Graphing data 537
  - 15.6.1 Reading a file 539
  - 15.6.2 General layout 541
  - 15.6.3 Scaling data 542
  - 15.6.4 Building the graph 543

## Chapter 16 Graphical User Interfaces 551

- 16.1 User interface alternatives 552
- 16.2 The “Next” button 553
- 16.3 A simple window 554
  - 16.3.1 A callback function 556
  - 16.3.2 A wait loop 559
  - 16.3.3 A lambda expression as a callback 560
- 16.4 **Button** and other **Widgets** 561
  - 16.4.1 **Widgets** 561
  - 16.4.2 **Buttons** 563
  - 16.4.3 **In\_box** and **Out\_box** 563
  - 16.4.4 **Menus** 564
- 16.5 An example 565

- 16.6 Control inversion 569
- 16.7 Adding a menu 570
- 16.8 Debugging GUI code 575

## Part III Data and Algorithms 581

### Chapter 17 Vector and Free Store 583

- 17.1 Introduction 584
- 17.2 `vector` basics 586
- 17.3 Memory, addresses, and pointers 588
  - 17.3.1 The `sizeof` operator 590
- 17.4 Free store and pointers 591
  - 17.4.1 Free-store allocation 593
  - 17.4.2 Access through pointers 594
  - 17.4.3 Ranges 595
  - 17.4.4 Initialization 596
  - 17.4.5 The null pointer 598
  - 17.4.6 Free-store deallocation 598
- 17.5 Destructors 601
  - 17.5.1 Generated destructors 603
  - 17.5.2 Destructors and free store 604
- 17.6 Access to elements 605
- 17.7 Pointers to class objects 606
- 17.8 Messing with types: `void*` and casts 608
- 17.9 Pointers and references 610
  - 17.9.1 Pointer and reference parameters 611
  - 17.9.2 Pointers, references, and inheritance 612
  - 17.9.3 An example: lists 613
  - 17.9.4 List operations 615
  - 17.9.5 List use 616
- 17.10 The `this` pointer 618
  - 17.10.1 More link use 620

### Chapter 18 Vectors and Arrays 627

- 18.1 Introduction 628
- 18.2 Initialization 629
- 18.3 Copying 631
  - 18.3.1 Copy constructors 633
  - 18.3.2 Copy assignments 634
  - 18.3.3 Copy terminology 636
  - 18.3.4 Moving 637

18.4	Essential operations	640
18.4.1	Explicit constructors	642
18.4.2	Debugging constructors and destructors	643
18.5	Access to <code>vector</code> elements	646
18.5.1	Overloading on <code>const</code>	647
18.6	Arrays	648
18.6.1	Pointers to array elements	650
18.6.2	Pointers and arrays	652
18.6.3	Array initialization	654
18.6.4	Pointer problems	656
18.7	Examples: palindrome	659
18.7.1	Palindromes using <code>string</code>	659
18.7.2	Palindromes using arrays	660
18.7.3	Palindromes using pointers	661

## Chapter 19 Vector, Templates, and Exceptions 667

19.1	The problems	668
19.2	Changing size	671
19.2.1	Representation	671
19.2.2	<code>reserve</code> and <code>capacity</code>	673
19.2.3	<code>resize</code>	674
19.2.4	<code>push_back</code>	674
19.2.5	Assignment	675
19.2.6	Our <code>vector</code> so far	677
19.3	Templates	678
19.3.1	Types as template parameters	679
19.3.2	Generic programming	681
19.3.3	Concepts	683
19.3.4	Containers and inheritance	686
19.3.5	Integers as template parameters	687
19.3.6	Template argument deduction	689
19.3.7	Generalizing <code>vector</code>	690
19.4	Range checking and exceptions	693
19.4.1	An aside: design considerations	694
19.4.2	A confession: macros	696
19.5	Resources and exceptions	697
19.5.1	Potential resource management problems	698
19.5.2	Resource acquisition is initialization	700
19.5.3	Guarantees	701
19.5.4	<code>unique_ptr</code>	703
19.5.5	Return by moving	704
19.5.6	RAII for <code>vector</code>	705

**Chapter 20 Containers and Iterators 711**

- 20.1 Storing and processing data 712
  - 20.1.1 Working with data 713
  - 20.1.2 Generalizing code 714
- 20.2 STL ideals 717
- 20.3 Sequences and iterators 720
  - 20.3.1 Back to the example 723
- 20.4 Linked lists 724
  - 20.4.1 List operations 726
  - 20.4.2 Iteration 727
- 20.5 Generalizing `vector` yet again 729
  - 20.5.1 Container traversal 732
  - 20.5.2 `auto` 732
- 20.6 An example: a simple text editor 734
  - 20.6.1 Lines 736
  - 20.6.2 Iteration 737
- 20.7 `vector`, `list`, and `string` 741
  - 20.7.1 `insert` and `erase` 742
- 20.8 Adapting our `vector` to the STL 745
- 20.9 Adapting built-in arrays to the STL 747
- 20.10 Container overview 749
  - 20.10.1 Iterator categories 751

**Chapter 21 Algorithms and Maps 757**

- 21.1 Standard library algorithms 758
- 21.2 The simplest algorithm: `find()` 759
  - 21.2.1 Some generic uses 761
- 21.3 The general search: `find_if()` 763
- 21.4 Function objects 765
  - 21.4.1 An abstract view of function objects 766
  - 21.4.2 Predicates on class members 767
  - 21.4.3 Lambda expressions 769
- 21.5 Numerical algorithms 770
  - 21.5.1 Accumulate 770
  - 21.5.2 Generalizing `accumulate()` 772
  - 21.5.3 `Inner product` 774
  - 21.5.4 Generalizing `inner_product()` 775
- 21.6 Associative containers 776
  - 21.6.1 `map` 776
  - 21.6.2 `map` overview 779
  - 21.6.3 Another `map` example 782
  - 21.6.4 `unordered_map` 785
  - 21.6.5 `set` 787

21.7	Copying	789
21.7.1	Copy	789
21.7.2	Stream iterators	790
21.7.3	Using a <code>set</code> to keep order	793
21.7.4	<code>copy_if</code>	794
21.8	Sorting and searching	794
21.9	Container algorithms	797

## Part IV Broadening the View 803

### Chapter 22 Ideals and History 805

22.1	History, ideals, and professionalism	806
22.1.1	Programming language aims and philosophies	807
22.1.2	Programming ideals	808
22.1.3	Styles/paradigms	815
22.2	Programming language history overview	818
22.2.1	The earliest languages	819
22.2.2	The roots of modern languages	821
22.2.3	The Algol family	826
22.2.4	Simula	833
22.2.5	C	836
22.2.6	C++	839
22.2.7	Today	842
22.2.8	Information sources	844

### Chapter 23 Text Manipulation 849

23.1	Text	850
23.2	Strings	850
23.3	I/O streams	855
23.4	Maps	855
23.4.1	Implementation details	861
23.5	A problem	864
23.6	The idea of regular expressions	866
23.6.1	Raw string literals	868
23.7	Searching with regular expressions	869
23.8	Regular expression syntax	872
23.8.1	Characters and special characters	872
23.8.2	Character classes	873
23.8.3	Repeats	874
23.8.4	Grouping	876
23.8.5	Alternation	876
23.8.6	Character sets and ranges	877
23.8.7	Regular expression errors	878

23.9	Matching with regular expressions	880
23.10	References	885

## Chapter 24 Numerics 889

24.1	Introduction	890
24.2	Size, precision, and overflow	890
24.2.1	Numeric limits	894
24.3	Arrays	895
24.4	C-style multidimensional arrays	896
24.5	The <b>Matrix</b> library	897
24.5.1	Dimensions and access	898
24.5.2	1D <b>Matrix</b>	901
24.5.3	2D <b>Matrix</b>	904
24.5.4	<b>Matrix</b> I/O	907
24.5.5	3D <b>Matrix</b>	907
24.6	An example: solving linear equations	908
24.6.1	Classical Gaussian elimination	910
24.6.2	Pivoting	911
24.6.3	Testing	912
24.7	Random numbers	914
24.8	The standard mathematical functions	917
24.9	Complex numbers	919
24.10	References	920

## Chapter 25 Embedded Systems Programming 925

25.1	Embedded systems	926
25.2	Basic concepts	929
25.2.1	Predictability	932
25.2.2	Ideals	932
25.2.3	Living with failure	933
25.3	Memory management	935
25.3.1	Free-store problems	936
25.3.2	Alternatives to the general free store	939
25.3.3	Pool example	940
25.3.4	Stack example	942
25.4	Addresses, pointers, and arrays	943
25.4.1	Unchecked conversions	943
25.4.2	A problem: dysfunctional interfaces	944
25.4.3	A solution: an interface class	947
25.4.4	Inheritance and containers	951
25.5	Bits, bytes, and words	954
25.5.1	Bits and bit operations	955
25.5.2	<b>bitset</b>	959

25.5.3	Signed and unsigned	961
25.5.4	Bit manipulation	965
25.5.5	Bitfields	967
25.5.6	An example: simple encryption	969
25.6	Coding standards	974
25.6.1	What should a coding standard be?	975
25.6.2	Sample rules	977
25.6.3	Real coding standards	983

## Chapter 26 Testing 989

26.1	What we want	990
26.1.1	Caveat	991
26.2	Proofs	992
26.3	Testing	992
26.3.1	Regression tests	993
26.3.2	Unit tests	994
26.3.3	Algorithms and non-algorithms	1001
26.3.4	System tests	1009
26.3.5	Finding assumptions that do not hold	1009
26.4	Design for testing	1011
26.5	Debugging	1012
26.6	Performance	1012
26.6.1	Timing	1015
26.7	References	1016

## Chapter 27 The C Programming Language 1021

27.1	C and C++: siblings	1022
27.1.1	C/C++ compatibility	1024
27.1.2	C++ features missing from C	1025
27.1.3	The C standard library	1027
27.2	Functions	1028
27.2.1	No function name overloading	1028
27.2.2	Function argument type checking	1029
27.2.3	Function definitions	1031
27.2.4	Calling C from C++ and C++ from C	1032
27.2.5	Pointers to functions	1034
27.3	Minor language differences	1036
27.3.1	<b>struct</b> tag namespace	1036
27.3.2	Keywords	1037
27.3.3	Definitions	1038
27.3.4	C-style casts	1040

27.3.5	Conversion of <code>void*</code>	1041
27.3.6	<code>enum</code>	1042
27.3.7	Namespaces	1042
27.4	Free store	1043
27.5	C-style strings	1045
27.5.1	C-style strings and <code>const</code>	1047
27.5.2	Byte operations	1048
27.5.3	An example: <code>strcpy()</code>	1049
27.5.4	A style issue	1049
27.6	Input/output: stdio	1050
27.6.1	Output	1050
27.6.2	Input	1052
27.6.3	Files	1053
27.7	Constants and macros	1054
27.8	Macros	1055
27.8.1	Function-like macros	1056
27.8.2	Syntax macros	1058
27.8.3	Conditional compilation	1058
27.9	An example: intrusive containers	1059

## Part V Appendices 1071

### Appendix A Language Summary 1073

A.1	General	1074
A.1.1	Terminology	1075
A.1.2	Program start and termination	1075
A.1.3	Comments	1076
A.2	Literals	1077
A.2.1	Integer literals	1077
A.2.2	Floating-point-literals	1079
A.2.3	Boolean literals	1079
A.2.4	Character literals	1079
A.2.5	String literals	1080
A.2.6	The pointer literal	1081
A.3	Identifiers	1081
A.3.1	Keywords	1081
A.4	Scope, storage class, and lifetime	1082
A.4.1	Scope	1082
A.4.2	Storage class	1083
A.4.3	Lifetime	1085

A.5	Expressions	1086
A.5.1	User-defined operators	1091
A.5.2	Implicit type conversion	1091
A.5.3	Constant expressions	1093
A.5.4	<b>sizeof</b>	1093
A.5.5	Logical expressions	1094
A.5.6	<b>new</b> and <b>delete</b>	1094
A.5.7	Casts	1095
A.6	Statements	1096
A.7	Declarations	1098
A.7.1	Definitions	1098
A.8	Built-in types	1099
A.8.1	Pointers	1100
A.8.2	Arrays	1101
A.8.3	References	1102
A.9	Functions	1103
A.9.1	Overload resolution	1104
A.9.2	Default arguments	1105
A.9.3	Unspecified arguments	1105
A.9.4	Linkage specifications	1106
A.10	User-defined types	1106
A.10.1	Operator overloading	1107
A.11	Enumerations	1107
A.12	Classes	1108
A.12.1	Member access	1108
A.12.2	Class member definitions	1112
A.12.3	Construction, destruction, and copy	1112
A.12.4	Derived classes	1116
A.12.5	Bitfields	1120
A.12.6	Unions	1121
A.13	Templates	1121
A.13.1	Template arguments	1122
A.13.2	Template instantiation	1123
A.13.3	Template member types	1124
A.14	Exceptions	1125
A.15	Namespaces	1127
A.16	Aliases	1128
A.17	Preprocessor directives	1128
A.17.1	<b>#include</b>	1128
A.17.2	<b>#define</b>	1129

## Appendix B Standard Library Summary 1131

B.1	Overview	1132
B.1.1	Header files	1133
B.1.2	Namespace <code>std</code>	1136
B.1.3	Description style	1136
B.2	Error handling	1137
B.2.1	Exceptions	1138
B.3	Iterators	1139
B.3.1	Iterator model	1140
B.3.2	Iterator categories	1142
B.4	Containers	1144
B.4.1	Overview	1146
B.4.2	Member types	1147
B.4.3	Constructors, destructors, and assignments	1148
B.4.4	Iterators	1148
B.4.5	Element access	1149
B.4.6	Stack and queue operations	1149
B.4.7	List operations	1150
B.4.8	Size and capacity	1150
B.4.9	Other operations	1151
B.4.10	Associative container operations	1151
B.5	Algorithms	1152
B.5.1	Nonmodifying sequence algorithms	1153
B.5.2	Modifying sequence algorithms	1154
B.5.3	Utility algorithms	1156
B.5.4	Sorting and searching	1157
B.5.5	Set algorithms	1159
B.5.6	Heaps	1160
B.5.7	Permutations	1160
B.5.8	<code>min</code> and <code>max</code>	1161
B.6	STL utilities	1162
B.6.1	Inserter	1162
B.6.2	Function objects	1163
B.6.3	<code>pair</code> and <code>tuple</code>	1165
B.6.4	<code>initializer_list</code>	1166
B.6.5	Resource management pointers	1167
B.7	I/O streams	1168
B.7.1	I/O streams hierarchy	1170
B.7.2	Error handling	1171
B.7.3	Input operations	1172

B.7.4	Output operations	1173
B.7.5	Formatting	1173
B.7.6	Standard manipulators	1173
B.8	String manipulation	1175
B.8.1	Character classification	1175
B.8.2	String	1176
B.8.3	Regular expression matching	1177
B.9	Numerics	1180
B.9.1	Numerical limits	1180
B.9.2	Standard mathematical functions	1181
B.9.3	Complex	1182
B.9.4	<b>valarray</b>	1183
B.9.5	Generalized numerical algorithms	1183
B.9.6	Random numbers	1184
B.10	Time	1185
B.11	C standard library functions	1185
B.11.1	Files	1186
B.11.2	The <b>printf()</b> family	1186
B.11.3	C-style strings	1191
B.11.4	Memory	1192
B.11.5	Date and time	1193
B.10.6	Etc.	1194
B.12	Other libraries	1195

## Appendix C Getting Started with Visual Studio 1197

C.1	Getting a program to run	1198
C.2	Installing Visual Studio	1198
C.3	Creating and running a program	1199
C.3.1	Create a new project	1199
C.3.2	Use the <b>std_lib_facilities.h</b> header file	1199
C.3.3	Add a C++ source file to the project	1200
C.3.4	Enter your source code	1200
C.3.5	Build an executable program	1200
C.3.6	Execute the program	1201
C.3.7	Save the program	1201
C.4	Later	1201

## Appendix D Installing FLTK 1203

D.1	Introduction	1204
D.2	Downloading FLTK	1204
D.3	Installing FLTK	1205
D.4	Using FLTK in Visual Studio	1205
D.5	Testing if it all worked	1206

**Appendix E GUI Implementation 1207**

- E.1 Callback implementation 1208
- E.2 **Widget** implementation 1209
- E.3 **Window** implementation 1210
- E.4 **Vector\_ref** 1212
- E.5 An example: manipulating **Widgets** 1213

*Glossary* 1217*Bibliography* 1223*Index* 1227

*This page intentionally left blank*

# Preface

“Damn the torpedoes!  
Full speed ahead.”

—Admiral Farragut

Programming is the art of expressing solutions to problems so that a computer can execute those solutions. Much of the effort in programming is spent finding and refining solutions. Often, a problem is only fully understood through the process of programming a solution for it.

This book is for someone who has never programmed before but is willing to work hard to learn. It helps you understand the principles and acquire the practical skills of programming using the C++ programming language. My aim is for you to gain sufficient knowledge and experience to perform simple useful programming tasks using the best up-to-date techniques. How long will that take? As part of a first-year university course, you can work through this book in a semester (assuming that you have a workload of four courses of average difficulty). If you work by yourself, don’t expect to spend less time than that (maybe 15 hours a week for 14 weeks).

Three months may seem a long time, but there’s a lot to learn and you’ll be writing your first simple programs after about an hour. Also, all learning is gradual: each chapter introduces new useful concepts and illustrates them with examples inspired by real-world uses. Your ability to express ideas in code – getting a computer to do what you want it to do – gradually and steadily increases as you go along. I never say, “Learn a month’s worth of theory and then see if you can use it.”

Why would you want to program? Our civilization runs on software. Without understanding software you are reduced to believing in “magic” and will be locked out of many of the most interesting, profitable, and socially useful technical fields of work. When I talk about programming, I think of the whole spectrum of computer programs from personal computer applications with GUIs (graphical user interfaces), through engineering calculations and embedded systems control applications (such as digital cameras, cars, and cell phones), to text manipulation applications as found in many humanities and business applications. Like mathematics, programming – when done well – is a valuable intellectual exercise that sharpens our ability to think. However, thanks to feedback from the computer, programming is more concrete than most forms of math, and therefore accessible to more people. It is a way to reach out and change the world – ideally for the better. Finally, programming can be great fun.

Why C++? You can’t learn to program without a programming language, and C++ directly supports the key concepts and techniques used in real-world software. C++ is one of the most widely used programming languages, found in an unsurpassed range of application areas. You find C++ applications everywhere from the bottom of the oceans to the surface of Mars. C++ is precisely and comprehensively defined by a nonproprietary international standard. Quality and/or free implementations are available on every kind of computer. Most of the programming concepts that you will learn using C++ can be used directly in other languages, such as C, C#, Fortran, and Java. Finally, I simply like C++ as a language for writing elegant and efficient code.

This is not the easiest book on beginning programming; it is not meant to be. I just aim for it to be the easiest book from which you can learn the basics of real-world programming. That’s quite an ambitious goal because much modern software relies on techniques considered advanced just a few years ago.

My fundamental assumption is that you want to write programs for the use of others, and to do so responsibly, providing a decent level of system quality; that is, I assume that you want to achieve a level of professionalism. Consequently, I chose the topics for this book to cover what is needed to get started with real-world programming, not just what is easy to teach and learn. If you need a technique to get basic work done right, I describe it, demonstrate concepts and language facilities needed to support the technique, provide exercises for it, and expect you to work on those exercises. If you just want to understand toy programs, you can get along with far less than I present. On the other hand, I won’t waste your time with material of marginal practical importance. If an idea is explained here, it’s because you’ll almost certainly need it.

If your desire is to use the work of others without understanding how things are done and without adding significantly to the code yourself, this book is not for you. If so, please consider whether you would be better served by another book and another language. If that is approximately your view of programming, please

also consider from where you got that view and whether it in fact is adequate for your needs. People often underestimate the complexity of programming as well as its value. I would hate for you to acquire a dislike for programming because of a mismatch between what you need and the part of the software reality I describe. There are many parts of the “information technology” world that do not require knowledge of programming. This book is aimed to serve those who do want to write or understand nontrivial programs.

Because of its structure and practical aims, this book can also be used as a second book on programming for someone who already knows a bit of C++ or for someone who programs in another language and wants to learn C++. If you fit into one of those categories, I refrain from guessing how long it will take you to read this book, but I do encourage you to do many of the exercises. This will help you to counteract the common problem of writing programs in older, familiar styles rather than adopting newer techniques where these are more appropriate. If you have learned C++ in one of the more traditional ways, you’ll find something surprising and useful before you reach Chapter 7. Unless your name is Stroustrup, what I discuss here is not “your father’s C++.”

Programming is learned by writing programs. In this, programming is similar to other endeavors with a practical component. You cannot learn to swim, to play a musical instrument, or to drive a car just from reading a book – you must practice. Nor can you learn to program without reading and writing lots of code. This book focuses on code examples closely tied to explanatory text and diagrams. You need those to understand the ideals, concepts, and principles of programming and to master the language constructs used to express them. That’s essential, but by itself, it will not give you the practical skills of programming. For that, you need to do the exercises and get used to the tools for writing, compiling, and running programs. You need to make your own mistakes and learn to correct them. There is no substitute for writing code. Besides, that’s where the fun is!

On the other hand, there is more to programming – much more – than following a few rules and reading the manual. This book is emphatically not focused on “the syntax of C++.” Understanding the fundamental ideals, principles, and techniques is the essence of a good programmer. Only well-designed code has a chance of becoming part of a correct, reliable, and maintainable system. Also, “the fundamentals” are what last: they will still be essential after today’s languages and tools have evolved or been replaced.

What about computer science, software engineering, information technology, etc.? Is that all programming? Of course not! Programming is one of the fundamental topics that underlie everything in computer-related fields, and it has a natural place in a balanced course of computer science. I provide brief introductions to key concepts and techniques of algorithms, data structures, user interfaces, data processing, and software engineering. However, this book is not a substitute for a thorough and balanced study of those topics.

Code can be beautiful as well as useful. This book is written to help you see that, to understand what it means for code to be beautiful, and to help you to master the principles and acquire the practical skills to create such code. Good luck with programming!

## A note to students

Of the many thousands of first-year students we have taught so far using this book at Texas A&M University, about 60% had programmed before and about 40% had never seen a line of code in their lives. Most succeeded, so you can do it, too.

You don't have to read this book as part of a course. The book is widely used for self-study. However, whether you work your way through as part of a course or independently, try to work with others. Programming has an – unfair – reputation as a lonely activity. Most people work better and learn faster when they are part of a group with a common aim. Learning together and discussing problems with friends is not cheating! It is the most efficient – as well as most pleasant – way of making progress. If nothing else, working with friends forces you to articulate your ideas, which is just about the most efficient way of testing your understanding and making sure you remember. You don't actually have to personally discover the answer to every obscure language and programming environment problem. However, please don't cheat yourself by not doing the drills and a fair number of exercises (even if no teacher forces you to do them). Remember: programming is (among other things) a practical skill that you need to practice to master. If you don't write code (do several exercises for each chapter), reading this book will be a pointless theoretical exercise.

Most students – especially thoughtful good students – face times when they wonder whether their hard work is worthwhile. When (not if) this happens to you, take a break, reread this Preface, and look at Chapter 1 (“Computers, People, and Programming”) and Chapter 22 (“Ideals and History”). There, I try to articulate what I find exciting about programming and why I consider it a crucial tool for making a positive contribution to the world. If you wonder about my teaching philosophy and general approach, have a look at Chapter 0 (“Notes to the Reader”).

You might find the weight of this book worrying, but it should reassure you that part of the reason for the heft is that I prefer to repeat an explanation or add an example rather than have you search for the one and only explanation. The other major reason is that the second half of the book is reference material and “additional material” presented for you to explore only if you are interested in more information about a specific area of programming, such as embedded systems programming, text analysis, or numerical computation.

And please don't be too impatient. Learning any major new and valuable skill takes time and is worth it.

## A note to teachers

No. This is not a traditional Computer Science 101 course. It is a book about how to construct working software. As such, it leaves out much of what a computer science student is traditionally exposed to (Turing completeness, state machines, discrete math, Chomsky grammars, etc.). Even hardware is ignored on the assumption that students have used computers in various ways since kindergarten. This book does not even try to mention most important CS topics. It is about programming (or more generally about how to develop software), and as such it goes into more detail about fewer topics than many traditional courses. It tries to do just one thing well, and computer science is not a one-course topic. If this book/course is used as part of a computer science, computer engineering, electrical engineering (many of our first students were EE majors), information science, or whatever program, I expect it to be taught alongside other courses as part of a well-rounded introduction.

Please read Chapter 0 (“Notes to the Reader”) for an explanation of my teaching philosophy, general approach, etc. Please try to convey those ideas to your students along the way.

## ISO standard C++

C++ is defined by an ISO standard. The first ISO C++ standard was ratified in 1998, so that version of C++ is known as C++98. I wrote the first edition of this book while working on the design of C++11. It was most frustrating not to be able to use the novel features (such as uniform initialization, range-**for**-loops, move semantics, lambdas, and concepts) to simplify the presentation of principles and techniques. However, the book was designed with C++11 in mind, so it was relatively easy to “drop in” the features in the contexts where they belonged. As of this writing, the current standard is C++11 from 2011, and facilities from the upcoming 2014 ISO standard, C++14, are finding their way into mainstream C++ implementations. The language used in this book is C++11 with a few C++14 features. For example, if your compiler complains about

```
vector<int> v1;
vector<int> v2 {v1}; // C++14-style copy construction
```

use

```
vector<int> v1;
vector<int> v2 = v1; // C++98-style copy construction
```

instead.

If your compiler does not support C++11, get a new compiler. Good, modern C++ compilers can be downloaded from a variety of suppliers; see [www.stroustrup.com/compilers.html](http://www.stroustrup.com/compilers.html). Learning to program using an earlier and less supportive version of the language can be unnecessarily hard.

## Support

The book's support website, [www.stroustrup.com/Programming](http://www.stroustrup.com/Programming), contains a variety of material supporting the teaching and learning of programming using this book. The material is likely to be improved with time, but for starters, you can find

- Slides for lectures based on the book
- An instructor's guide
- Header files and implementations of libraries used in the book
- Code for examples in the book
- Solutions to selected exercises
- Potentially useful links
- Errata

Suggestions for improvements are always welcome.

## Acknowledgments

I'd especially like to thank my late colleague and co-teacher Lawrence "Pete" Petersen for encouraging me to tackle the task of teaching beginners long before I'd otherwise have felt comfortable doing that, and for supplying the practical teaching experience to make the course succeed. Without him, the first version of the course would have been a failure. We worked together on the first versions of the course for which this book was designed and together taught it repeatedly, learning from our experiences, improving the course and the book. My use of "we" in this book initially meant "Pete and me."

Thanks to the students, teaching assistants, and peer teachers of ENGR 112, ENGR 113, and CSCE 121 at Texas A&M University who directly and indirectly helped us construct this book, and to Walter Daugherty, Hyunyoung Lee, Teresa Leyk, Ronnie Ward, and Jennifer Welch, who have also taught the course. Also thanks to Damian Dechev, Tracy Hammond, Arne Tolstrup Madsen, Gabriel Dos Reis, Nicholas Stroustrup, J. C. van Winkel, Greg Versoender, Ronnie Ward, and Leor Zolman for constructive comments on drafts of this book. Thanks to Mogens Hansen for explaining about engine control software. Thanks to Al Aho, Stephen Edwards, Brian Kernighan, and Daisy Nguyen for helping me hide away from distractions to get writing done during the summers.

Thanks to Art Werschulz for many constructive comments based on his use of the first edition of this book in courses at Fordham University in New York City and to Nick Maclaren for many detailed comments on the exercises based on his use of the first edition of this book at Cambridge University. His students had dramatically different backgrounds and professional needs from the TAMU first-year students.

Thanks to the reviewers that Addison-Wesley found for me. Their comments, mostly based on teaching either C++ or Computer Science 101 at the college level, have been invaluable: Richard Enbody, David Gustafson, Ron McCarty, and K. Narayanaswamy. Also thanks to my editor, Peter Gordon, for many useful comments and (not least) for his patience. I'm very grateful to the production team assembled by Addison-Wesley; they added much to the quality of the book: Linda Begley (proofreader), Kim Arney (compositor), Rob Mauhar (illustrator), Julie Nahil (production editor), and Barbara Wood (copy editor).

Thanks to the translators of the first edition, who found many problems and helped clarify many points. In particular, Loïc Joly and Michel Michaud did a thorough technical review of the French translation that led to many improvements.

I would also like to thank Brian Kernighan and Doug McIlroy for setting a very high standard for writing about programming, and Dennis Ritchie and Kristen Nygaard for providing valuable lessons in practical language design.

*This page intentionally left blank*



## Notes to the Reader

“When the terrain disagrees with the map, trust the terrain.”

—Swiss army proverb

This chapter is a grab bag of information; it aims to give you an idea of what to expect from the rest of the book. Please skim through it and read what you find interesting. A teacher will find most parts immediately useful. If you are reading this book without the benefit of a good teacher, please don't try to read and understand everything in this chapter; just look at “The structure of this book” and the first part of the “A philosophy of teaching and learning” sections. You may want to return and reread this chapter once you feel comfortable writing and executing small programs.

**0.1 The structure of this book**

- 0.1.1 General approach
- 0.1.2 Drills, exercises, etc.
- 0.1.3 What comes after this book?

**0.2 A philosophy of teaching and learning**

- 0.2.1 The order of topics
- 0.2.2 Programming and programming language
- 0.2.3 Portability

**0.3 Programming and computer science****0.4 Creativity and problem solving****0.5 Request for feedback****0.6 References****0.7 Biographies**

## 0.1 The structure of this book

This book consists of four parts and a collection of appendices:

- *Part I, “The Basics,”* presents the fundamental concepts and techniques of programming together with the C++ language and library facilities needed to get started writing code. This includes the type system, arithmetic operations, control structures, error handling, and the design, implementation, and use of functions and user-defined types.
- *Part II, “Input and Output,”* describes how to get numeric and text data from the keyboard and from files, and how to produce corresponding output to the screen and to files. Then, it shows how to present numeric data, text, and geometric shapes as graphical output, and how to get input into a program from a graphical user interface (GUI).
- *Part III, “Data and Algorithms,”* focuses on the C++ standard library’s containers and algorithms framework (the STL, standard template library). It shows how containers (such as `vector`, `list`, and `map`) are implemented (using pointers, arrays, dynamic memory, exceptions, and templates) and used. It also demonstrates the design and use of standard library algorithms (such as `sort`, `find`, and `inner_product`).
- *Part IV, “Broadening the View,”* offers a perspective on programming through a discussion of ideals and history, through examples (such as matrix computation, text manipulation, testing, and embedded systems programming), and through a brief description of the C language.
- *Appendices* provide useful information that doesn’t fit into a tutorial presentation, such as surveys of C++ language and standard library facilities, and descriptions of how to get started with an integrated development environment (IDE) and a graphical user interface (GUI) library.

Unfortunately, the world of programming doesn't really fall into four cleanly separated parts. Therefore, the “parts” of this book provide only a coarse classification of topics. We consider it a useful classification (obviously, or we wouldn't have used it), but reality has a way of escaping neat classifications. For example, we need to use input operations far sooner than we can give a thorough explanation of C++ standard I/O streams (input/output streams). Where the set of topics needed to present an idea conflicts with the overall classification, we explain the minimum needed for a good presentation, rather than just referring to the complete explanation elsewhere. Rigid classifications work much better for manuals than for tutorials.

The order of topics is determined by programming techniques, rather than programming language features; see §0.2. For a presentation organized around language features, see Appendix A.

To ease review and to help you if you miss a key point during a first reading where you have yet to discover which kind of information is crucial, we place three kinds of “alert markers” in the margin:

- Blue: concepts and techniques (this paragraph is an example of that)
- Green: advice
- Red: warning

### 0.1.1 General approach

In this book, we address you directly. That is simpler and clearer than the conventional “professional” indirect form of address, as found in most scientific papers. By “you” we mean “you, the reader,” and by “we” we refer either to “ourselves, the author and teachers,” or to you and us working together through a problem, as we might have done had we been in the same room.

This book is designed to be read chapter by chapter from the beginning to the end. Often, you'll want to go back to look at something a second or a third time. In fact, that's the only sensible approach, as you'll always dash past some details that you don't yet see the point in. In such cases, you'll eventually go back again. However, despite the index and the cross-references, this is not a book that you can open to any page and start reading with any expectation of success. Each section and each chapter assume understanding of what came before.

Each chapter is a reasonably self-contained unit, meant to be read in “one sitting” (logically, if not always feasible on a student's tight schedule). That's one major criterion for separating the text into chapters. Other criteria include that a chapter is a suitable unit for drills and exercises and that each chapter presents some specific concept, idea, or technique. This plurality of criteria has left a few chapters uncomfortably long, so please don't take “in one sitting” too literally. In particular, once you have thought about the review questions, done the drill, and

worked on a few exercises, you'll often find that you have to go back to reread a few sections and that several days have gone by. We have clustered the chapters into "parts" focused on a major topic, such as input/output. These parts make good units of review.

Common praise for a textbook is "It answered all my questions just as I thought of them!" That's an ideal for minor technical questions, and early readers have observed the phenomenon with this book. However, that cannot be the whole ideal. We raise questions that a novice would probably not think of. We aim to ask and answer questions that you need to consider when writing quality software for the use of others. Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer. Asking only the easy and obvious questions would make you feel good, but it wouldn't help make you a programmer.

We try to respect your intelligence and to be considerate about your time. In our presentation, we aim for professionalism rather than cuteness, and we'd rather understate a point than hype it. We try not to exaggerate the importance of a programming technique or a language feature, but please don't underestimate a simple statement like "This is often useful." If we quietly emphasize that something is important, we mean that you'll sooner or later waste days if you don't master it. Our use of humor is more limited than we would have preferred, but experience shows that people's ideas of what is funny differ dramatically and that a failed attempt at humor can be confusing.

We do not pretend that our ideas or the tools offered are perfect. No tool, library, language, or technique is "the solution" to all of the many challenges facing a programmer. At best, it can help you to develop and express your solution. We try hard to avoid "white lies"; that is, we refrain from oversimplified explanations that are clear and easy to understand, but not true in the context of real languages and real problems. On the other hand, this book is not a reference; for more precise and complete descriptions of C++, see Bjarne Stroustrup, *The C++ Programming Language, Fourth Edition* (Addison-Wesley, 2013), and the ISO C++ standard.

### 0.1.2 Drills, exercises, etc.

Programming is not just an intellectual activity, so writing programs is necessary to master programming skills. We provide two levels of programming practice:

- *Drills*: A drill is a very simple exercise devised to develop practical, almost mechanical skills. A drill usually consists of a sequence of modifications of a single program. You should do every drill. A drill is not asking for deep understanding, cleverness, or initiative. We consider the drills part of the basic fabric of the book. If you haven't done the drills, you have not "done" the book.

- *Exercises:* Some exercises are trivial and others are very hard, but most are intended to leave some scope for initiative and imagination. If you are serious, you'll do quite a few exercises. At least do enough to know which are difficult for you. Then do a few more of those. That's how you'll learn the most. The exercises are meant to be manageable without exceptional cleverness, rather than to be tricky puzzles. However, we hope that we have provided exercises that are hard enough to challenge anybody and enough exercises to exhaust even the best student's available time. We do not expect you to do them all, but feel free to try.

In addition, we recommend that you (every student) take part in a small project (and more if time allows for it). A project is intended to produce a complete useful program. Ideally, a project is done by a small group of people (e.g., three people) working together for about a month while working through the chapters in Part III. Most people find the projects the most fun and what ties everything together.

Some people like to put the book aside and try some examples before reading to the end of a chapter; others prefer to read ahead to the end before trying to get code to run. To support readers with the former preference, we provide simple suggestions for practical work labeled “**Try this**” at natural breaks in the text. A **Try this** is generally in the nature of a drill focused narrowly on the topic that precedes it. If you pass a **Try this** without trying – maybe because you are not near a computer or you find the text riveting – do return to it when you do the chapter drill; a **Try this** either complements the chapter drill or is a part of it.

At the end of each chapter you'll find a set of review questions. They are intended to point you to the key ideas explained in the chapter. One way to look at the review questions is as a complement to the exercises: the exercises focus on the practical aspects of programming, whereas the review questions try to help you articulate the ideas and concepts. In that, they resemble good interview questions.

The “Terms” section at the end of each chapter presents the basic vocabulary of programming and of C++. If you want to understand what people say about programming topics and to articulate your own ideas, you should know what each means.

Learning involves repetition. Our ideal is to make every important point at least twice and to reinforce it with exercises.

### 0.1.3 What comes after this book?

At the end of this book, will you be an expert at programming and at C++? Of course not! When done well, programming is a subtle, deep, and highly skilled art building on a variety of technical skills. You should no more expect to be an expert at programming in four months than you should expect to be an expert in biology, in math, in a natural language (such as Chinese, English, or Danish), or



at playing the violin in four months – or in half a year, or a year. What you should hope for, and what you can expect if you approach this book seriously, is to have a really good start that allows you to write relatively simple useful programs, to be able to read more complex programs, and to have a good conceptual and practical background for further work.

The best follow-up to this initial course is to work on a real project developing code to be used by someone else. After that, or (even better) in parallel with a real project, read either a professional-level general textbook (such as Stroustrup, *The C++ Programming Language*), a more specialized book relating to the needs of your project (such as Qt for GUI, or ACE for distributed programming), or a textbook focusing on a particular aspect of C++ (such as Koenig and Moo, *Accelerated C++*; Sutter's *Exceptional C++*; or Gamma et al., *Design Patterns*). For more references, see §0.6 or the Bibliography section at the back of the book.

Eventually, you should learn another programming language. We don't consider it possible to be a professional in the realm of software – even if you are not primarily a programmer – without knowing more than one language.

## 0.2 A philosophy of teaching and learning

What are we trying to help you learn? And how are we approaching the process of teaching? We try to present the minimal concepts, techniques, and tools for you to do effective practical programs, including

- Program organization
- Debugging and testing
- Class design
- Computation
- Function and algorithm design
- Graphics (two-dimensional only)
- Graphical user interfaces (GUIs)
- Text manipulation
- Regular expression matching
- Files and stream input and output (I/O)
- Memory management
- Scientific/numerical/engineering calculations
- Design and programming ideals
- The C++ standard library
- Software development strategies
- C-language programming techniques

Working our way through these topics, we cover the programming techniques called procedural programming (as with the C programming language), data abstraction, object-oriented programming, and generic programming. The main topic of this book is *programming*, that is, the ideals, techniques, and tools of expressing ideas in code. The C++ programming language is our main tool, so we describe many of C++’s facilities in some detail. But please remember that C++ is just a tool, rather than the main topic of this book. This is “programming using C++,” not “C++ with a bit of programming theory.”

Each topic we address serves at least two purposes: it presents a technique, concept, or principle and also a practical language or library feature. For example, we use the interface to a two-dimensional graphics system to illustrate the use of classes and inheritance. This allows us to be economical with space (and your time) and also to emphasize that programming is more than simply slinging code together to get a result as quickly as possible. The C++ standard library is a major source of such “double duty” examples – many even do triple duty. For example, we introduce the standard library `vector`, use it to illustrate widely useful design techniques, and show many of the programming techniques used to implement it. One of our aims is to show you how major library facilities are implemented and how they map to hardware. We insist that craftsmen must understand their tools, not just consider them “magical.”

Some topics will be of greater interest to some programmers than to others. However, we encourage you not to prejudge your needs (how would you know what you’ll need in the future?) and at least look at every chapter. If you read this book as part of a course, your teacher will guide your selection.

We characterize our approach as “depth-first.” It is also “concrete-first” and “concept-based.” First, we quickly (well, relatively quickly, Chapters 1–11) assemble a set of skills needed for writing small practical programs. In doing so, we present a lot of tools and techniques in minimal detail. We focus on simple concrete code examples because people grasp the concrete faster than the abstract. That’s simply the way most humans learn. At this initial stage, you should not expect to understand every little detail. In particular, you’ll find that trying something slightly different from what just worked can have “mysterious” effects. Do try, though! And please do the drills and exercises we provide. Just remember that early on you just don’t have the concepts and skills to accurately estimate what’s simple and what’s complicated; expect surprises and learn from them.

We move fast in this initial phase – we want to get you to the point where you can write interesting programs as fast as possible. Someone will argue, “We must move slowly and carefully; we must walk before we can run!” But have you ever watched a baby learning to walk? Babies really do run by themselves before they learn the finer skills of slow, controlled walking. Similarly, you will dash ahead, occasionally stumbling, to get a feel of programming before slowing down to gain the necessary finer control and understanding. You must run before you can walk!



It is essential that you don't get stuck in an attempt to learn "everything" about some language detail or technique. For example, you could memorize all of C++'s built-in types and all the rules for their use. Of course you could, and doing so might make you feel knowledgeable. However, it would not make you a programmer. Skipping details will get you "burned" occasionally for lack of knowledge, but it is the fastest way to gain the perspective needed to write good programs. Note that our approach is essentially the one used by children learning their native language and also the most effective approach used to teach foreign languages. We encourage you to seek help from teachers, friends, colleagues, instructors, Mentors, etc. on the inevitable occasions when you are stuck. Be assured that nothing in these early chapters is fundamentally difficult. However, much will be unfamiliar and might therefore feel difficult at first.

Later, we build on the initial skills to broaden your base of knowledge and skills. We use examples and exercises to solidify your understanding, and to provide a conceptual base for programming.



We place a heavy emphasis on ideals and reasons. You need ideals to guide you when you look for practical solutions – to know when a solution is good and principled. You need to understand the reasons behind those ideals to understand why they should be your ideals, why aiming for them will help you and the users of your code. Nobody should be satisfied with "because that's the way it is" as an explanation. More importantly, an understanding of ideals and reasons allows you to generalize from what you know to new situations and to combine ideas and tools in novel ways to address new problems. Knowing "why" is an essential part of acquiring programming skills. Conversely, just memorizing lots of poorly understood rules and language facilities is limiting, a source of errors, and a massive waste of time. We consider your time precious and try not to waste it.

Many C++ language-technical details are banished to appendices and manuals, where you can look them up when needed. We assume that you have the initiative to search out information when needed. Use the index and the table of contents. Don't forget the online help facilities of your compiler, and the web. Remember, though, to consider every web resource highly suspect until you have reason to believe better of it. Many an authoritative-looking website is put up by a programming novice or someone with something to sell. Others are simply outdated. We provide a collection of links and information on our support website: [www.stroustrup.com/Programming](http://www.stroustrup.com/Programming).

Please don't be too impatient for "realistic" examples. Our ideal example is the shortest and simplest code that directly illustrates a language facility, a concept, or a technique. Most real-world examples are far messier than ours, yet do not consist of more than a combination of what we demonstrate. Successful commercial programs with hundreds of thousands of lines of code are based on techniques that we illustrate in a dozen 50-line programs. The fastest way to understand real-world code is through a good understanding of the fundamentals.

On the other hand, we do not use “cute examples involving cuddly animals” to illustrate our points. We assume that you aim to write real programs to be used by real people, so every example that is not presented as language-technical is taken from a real-world use. Our basic tone is that of professionals addressing (future) professionals.

### 0.2.1 The order of topics

There are many ways to teach people how to program. Clearly, we don’t subscribe to the popular “the way I learned to program is the best way to learn” theories. To ease learning, we early on present topics that would have been considered advanced only a few years ago. Our ideal is for the topics we present to be driven by problems you meet as you learn to program, to flow smoothly from topic to topic as you increase your understanding and practical skills. The major flow of this book is more like a story than a dictionary or a hierarchical order.

It is impossible to learn all the principles, techniques, and language facilities needed to write a program at once. Consequently, we have to choose a subset of principles, techniques, and features to start with. More generally, a textbook or a course must lead students through a series of subsets. We consider it our responsibility to select topics and to provide emphasis. We can’t just present everything, so we must choose; what we leave out is at least as important as what we leave in – at each stage of the journey.

For contrast, it may be useful for you to see a list of (severely abbreviated) characterizations of approaches that we decided not to take:

- “*C first*”: This approach to learning C++ is wasteful of students’ time and leads to poor programming practices by forcing students to approach problems with fewer facilities, techniques, and libraries than necessary. C++ provides stronger type checking than C, a standard library with better support for novices, and exceptions for error handling.
- *Bottom-up*: This approach distracts from learning good and effective programming practices. By forcing students to solve problems with insufficient support from the language and libraries, it promotes poor and wasteful programming practices.
- “*If you present something, you must present it fully*”: This approach implies a bottom-up approach (by drilling deeper and deeper into every topic touched). It bores novices with technical details they have no interest in and quite likely will not need for years to come. Once you can program, you can look up technical details in a manual. Manuals are good at that, whereas they are awful for initial learning of concepts.
- *Top-down*: This approach, working from first principles toward details, tends to distract readers from the practical aspects of programming and

force them to concentrate on high-level concepts before they have any chance of appreciating their importance. For example, you simply can't appreciate proper software development principles before you have learned how easy it is to make a mistake in a program and how hard it can be to correct it.

- “*Abstract first*”: Focusing on general principles and protecting the student from nasty real-world constraints can lead to a disdain for real-world problems, languages, tools, and hardware constraints. Often, this approach is supported by “teaching languages” that cannot be used later and (deliberately) insulate students from hardware and system concerns.
- “*Software engineering principles first*”: This approach and the abstract-first approach tend to share the problems of the top-down approach: without concrete examples and practical experience, you simply cannot appreciate the value of abstraction and proper software development practices.
- “*Object-oriented from day one*”: Object-oriented programming is one of the best ways of organizing code and programming efforts, but it is not the only effective way. In particular, we feel that a grounding in the basics of types and algorithmic code is a prerequisite for appreciation of the design of classes and class hierarchies. We do use user-defined types (what some people would call “objects”) from day one, but we don’t show how to design a class until Chapter 6 and don’t show a class hierarchy until Chapter 12.
- “*Just believe in magic*”: This approach relies on demonstrations of powerful tools and techniques without introducing the novice to the underlying techniques and facilities. This leaves the student guessing – and usually guessing wrong – about why things are the way they are, what it costs to use them, and where they can be reasonably applied. This can lead to overrigid following of familiar patterns of work and become a barrier to further learning.

Naturally, we do not claim that these other approaches are never useful. In fact, we use several of these for specific subtopics where their strengths can be appreciated. However, as general approaches to learning programming aimed at real-world use, we reject them and apply our alternative: concrete-first and depth-first with an emphasis on concepts and techniques.

### 0.2.2 Programming and programming language

We teach programming first and treat our chosen programming language as secondary, as a tool. Our general approach can be used with any general-purpose programming language. Our primary aim is to help you learn general concepts,

principles, and techniques. However, those cannot be appreciated in isolation. For example, details of syntax, the kinds of ideas that can be directly expressed, and tool support differ from programming language to programming language. However, many of the fundamental techniques for producing bug-free code, such as writing logically simple code (Chapters 5 and 6), establishing invariants (§9.4.3), and separating interfaces from implementation details (§9.7 and §14.1–2), vary little from programming language to programming language.

Programming and design techniques must be learned using a programming language. Design, code organization, and debugging are not skills you can acquire in the abstract. You need to write code in some programming language and gain practical experience with that. This implies that you must learn the basics of a programming language. We say “the basics” because the days when you could learn all of a major industrial language in a few weeks are gone for good. The parts of C++ we present were chosen as the subset that most directly supports the production of good code. Also, we present C++ features that you can’t avoid encountering either because they are necessary for logical completeness or are common in the C++ community.

### 0.2.3 Portability

It is common to write C++ to run on a variety of machines. Major C++ applications run on machines we haven’t ever heard of! We consider portability and the use of a variety of machine architectures and operating systems most important. Essentially every example in this book is not only ISO Standard C++, but also portable. Unless specifically stated, the code we present should work on every C++ implementation and has been tested on several machines and operating systems.

The details of how to compile, link, and run a C++ program differ from system to system. It would be tedious to mention the details of every system and every compiler each time we need to refer to an implementation issue. In Appendix C, we give the most basic information about getting started using Visual Studio and Microsoft C++ on a Windows machine.

If you have trouble with one of the popular, but rather elaborate, IDEs (integrated development environments), we suggest you try working from the command line; it’s surprisingly simple. For example, here is the full set of commands needed to compile, link, and execute a simple program consisting of two source files, `my_file1.cpp` and `my_file2.cpp`, using the GNU C++ compiler on a Unix or Linux system:

```
c++ -o my_program my_file1.cpp my_file2.cpp  
./my_program
```

Yes, that really is all it takes.

## 0.3 Programming and computer science

Is programming all that there is to computer science? Of course not! The only reason we raise this question is that people have been known to be confused about this. We touch upon major topics from computer science, such as algorithms and data structures, but our aim is to teach programming: the design and implementation of programs. That is both more and less than most accepted notions of computer science:

- *More*, because programming involves many technical skills that are not usually considered part of any science
- *Less*, because we do not systematically present the foundation for the parts of computer science we use

The aim of this book is to be part of a course in computer science (if becoming a computer scientist is your aim), to be the foundation for the first of many courses in software construction and maintenance (if your aim is to become a programmer or a software engineer), and in general to be part of a greater whole.

We rely on computer science throughout and we emphasize principles, but we teach programming as a practical skill based on theory and experience, rather than as a science.

## 0.4 Creativity and problem solving

The primary aim of this book is to help you to express your ideas in code, not to teach you how to get those ideas. Along the way, we give many examples of how we can address a problem, usually through analysis of a problem followed by gradual refinement of a solution. We consider programming itself a form of problem solving: only through complete understanding of a problem and its solution can you express a correct program for it, and only through constructing and testing a program can you be certain that your understanding is complete. Thus, programming is inherently part of an effort to gain understanding. However, we aim to demonstrate this through examples, rather than through “preaching” or presentation of detailed prescriptions for problem solving.

## 0.5 Request for feedback

We don’t think that the perfect textbook can exist; the needs of individuals differ too much for that. However, we’d like to make this book and its supporting materials as good as we can make them. For that, we need feedback; a good textbook cannot be written in isolation from its readers. Please send us reports on errors, typos, unclear text, missing explanations, etc. We’d also appreciate suggestions

for better exercises, better examples, and topics to add, topics to delete, etc. Constructive comments will help future readers and we'll post errata on our support website: [www.stroustrup.com/Programming](http://www.stroustrup.com/Programming).

## 0.6 References

Along with listing the publications mentioned in this chapter, this section also includes publications you might find helpful.

- Becker, Pete, ed. *The C++ Standard*. ISO/IEC 14882:2011.
- Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4, Second Edition*. Prentice Hall, 2008. ISBN 0132354160.
- Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley, 2005. ISBN 0321334876.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2001. ISBN 0201604647.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2002. ISBN 0201795256.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
- Stroustrup, Bjarne. "Learning Standard C++ as a New Language." *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 0321563840.
- Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 0321958314.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 1999. ISBN 0201615622.

A more comprehensive list of references can be found in the Bibliography section at the back of the book.

## 0.7 Biographies

You might reasonably ask, "Who are these guys who want to teach me how to program?" So here is some biographical information. I, Bjarne Stroustrup, wrote this book, and together with Lawrence "Pete" Petersen, I designed and taught the university-level beginner's (first-year) course that was developed concurrently with the book, using drafts of the book.

## Bjarne Stroustrup



I'm the designer and original implementer of the C++ programming language. I have used the language, and many other programming languages, for a wide variety of programming tasks over the last 40 years or so. I just love elegant and efficient code used in challenging applications, such as robot control, graphics, games, text analysis, and networking. I have taught design, programming, and C++ to people of essentially all abilities and interests. I'm a founding member of the ISO standards committee for C++ where I serve as the chair of the working group for language evolution.

This is my first introductory book. My other books, such as *The C++ Programming Language* and *The Design and Evolution of C++*, were written for experienced programmers.

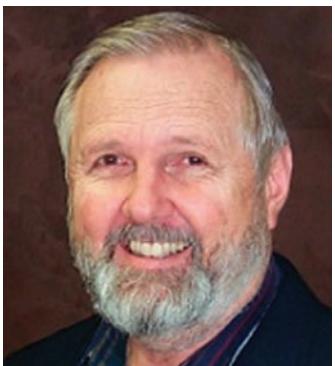
I was born into a blue-collar (working-class) family in Århus, Denmark, and got my master's degree in mathematics with computer science in my hometown university. My Ph.D. in computer science is from Cambridge University, England. I worked for AT&T for about 25 years, first in the famous Computer Science Research Center of Bell Labs – where Unix, C, C++, and so much more was invented – and later in AT&T Labs–Research.

I'm a member of the U.S. National Academy of Engineering, a Fellow of the ACM, and an IEEE Fellow. As the first computer scientist ever, I received the 2005 William Procter Prize for Scientific Achievement from Sigma Xi (the scientific research society). In 2010, I received the University of Århus's oldest and most prestigious honor for contributions to science by a person associated with the university, the *Rigmor og Carl Holst-Knudsens Videnskapspris*. In 2013, I was made Honorary Doctor of Computer Science from the National Research University, ITMO, St. Petersburg, Russia.

I do have a life outside work. I'm married and have two children, one a medical doctor and one a Post-doctoral Research Fellow. I read a lot (including history, science fiction, crime, and current affairs) and like most kinds of music (including classical, rock, blues, and country). Good food with friends is an essential part of life, and I enjoy visiting interesting places and people, all over the world. To be able to enjoy the good food, I run.

For more information, see my home pages: [www.stroustrup.com](http://www.stroustrup.com). In particular, there you can find out how to pronounce my name.

## Lawrence “Pete” Petersen



In late 2006, Pete introduced himself as follows: “I am a teacher. For almost 20 years, I have taught programming languages at Texas A&M. I have been selected by students for Teaching Excellence Awards five times and in 1996 received the Distinguished Teaching Award from the Alumni Association for the College of Engineering. I am a Fellow of the Wakonse Program for Teaching Excellence and a Fellow of the Academy for Educator Development.

“As the son of an army officer, I was raised on the move. After completing a degree in philosophy

at the University of Washington, I served in the army for 22 years as a Field Artillery Officer and as a Research Analyst for Operational Testing. I taught at the Field Artillery Officers’ Advanced Course at Fort Sill, Oklahoma, from 1971 to 1973. In 1979 I helped organize a Test Officers’ Training Course and taught it as lead instructor at nine different locations across the United States from 1978 to 1981 and from 1985 to 1989.

“In 1991 I formed a small software company that produced management software for university departments until 1999. My interests are in teaching, designing, and programming software that real people can use. I completed master’s degrees in industrial engineering at Georgia Tech and in education curriculum and instruction at Texas A&M. I also completed a master’s program in microcomputers from NTS. My Ph.D. is in information and operations management from Texas A&M.

“My wife, Barbara, and I live in Bryan, Texas. We like to travel, garden, and entertain; and we spend as much time as we can with our sons and their families, and especially with our grandchildren, Angelina, Carlos, Tess, Avery, Nicholas, and Jordan.”

Sadly, Pete died of lung cancer in 2007. Without him, the course would never have succeeded.

## Postscript

Most chapters provide a short “postscript” that attempts to give some perspective on the information presented in the chapter. We do that with the realization that the information can be – and often is – daunting and will only be fully comprehended after doing exercises, reading further chapters (which apply the ideas of the chapter), and a later review. *Don’t panic!* Relax; this is natural and expected. You won’t become an expert in a day, but you can become a reasonably competent programmer as you work your way through the book. On the way, you’ll encounter much information, many examples, and many techniques that lots of programmers have found stimulating and fun.

*This page intentionally left blank*



# Vectors and Arrays

“Caveat emptor!”

—Good advice

This chapter describes how vectors are copied and accessed through subscripting. To do that, we discuss copying in general and consider **vector**'s relation to the lower-level notion of arrays. We present arrays' relation to pointers and consider the problems arising from their use. We also present the five essential operations that must be considered for every type: construction, default construction, copy construction, copy assignment, and destruction. In addition, a container needs a move constructor and a move assignment.

<b>18.1 Introduction</b>	<b>18.6 Arrays</b>
<b>18.2 Initialization</b>	<b>18.6.1 Pointers to array elements</b>
<b>18.3 Copying</b>	<b>18.6.2 Pointers and arrays</b>
18.3.1 Copy constructors	<b>18.6.3 Array initialization</b>
18.3.2 Copy assignments	<b>18.6.4 Pointer problems</b>
18.3.3 Copy terminology	
18.3.4 Moving	
<b>18.4 Essential operations</b>	<b>18.7 Examples: palindrome</b>
18.4.1 Explicit constructors	18.7.1 Palindromes using <code>string</code>
18.4.2 Debugging constructors and destructors	18.7.2 Palindromes using arrays
<b>18.5 Access to <code>vector</code> elements</b>	18.7.3 Palindromes using pointers
18.5.1 Overloading on <code>const</code>	

## 18.1 Introduction

To get into the air, a plane has to accelerate along the runway until it moves fast enough to “jump” into the air. While the plane is lumbering along the runway, it is little more than a particularly heavy and awkward truck. Once in the air, it soars to become an altogether different, elegant, and efficient vehicle. It is in its true element.

In this chapter, we are in the middle of a “run” to gather enough programming language features and techniques to get away from the constraints and difficulties of plain computer memory. We want to get to the point where we can program using types that provide exactly the properties we want based on logical needs. To “get there” we have to overcome a number of fundamental constraints related to access to the bare machine, such as the following:

- An object in memory is of fixed size.
- An object in memory is in one specific place.
- The computer provides only a few fundamental operations on such objects (such as copying a word, adding the values from two words, etc.).

Basically, those are the constraints on the built-in types and operations of C++ (as inherited through C from hardware; see §22.2.5 and Chapter 27). In Chapter 17, we saw the beginnings of a `vector` type that controls all access to its elements and provides us with operations that seem “natural” from the point of view of a user, rather than from the point of view of hardware.

This chapter focuses on the notion of copying. This is an important but rather technical point: What do we mean by copying a nontrivial object? To what extent

are the copies independent after a copy operation? What copy operations are there? How do we specify them? And how do they relate to other fundamental operations, such as initialization and cleanup?

Inevitably, we get to discuss how memory is manipulated when we don't have higher-level types such as `vector` and `string`. We examine arrays and pointers, their relationship, their use, and the traps and pitfalls of their use. This is essential information to anyone who gets to work with low-level uses of C++ or C code.

Please note that the details of `vector` are peculiar to `vectors` and the C++ ways of building new higher-level types from lower-level ones. However, every "higher-level" type (`string`, `vector`, `list`, `map`, etc.) in every language is somehow built from the same machine primitives and reflects a variety of resolutions to the fundamental problems described here.

## 18.2 Initialization

Consider our `vector` as it was at the end of Chapter 17:

```
class vector {
    int sz;                      // the size
    double* elem;                // a pointer to the elements
public:
    vector(int s)                  // constructor
        :sz{s}, elem{new double[s]} { /* ... */ } // allocates memory
    ~vector()                     // destructor
        { delete[] elem; }           // deallocates memory
    // ...
};
```

That's fine, but what if we want to initialize a vector to a set of values that are not defaults? For example:

```
vector v1 = {1.2, 7.89, 12.34};
```

We can do that, and it is much better than initializing to default values and then assigning the values we really want:

```
vector v2(2);      // tedious and error-prone
v2[0] = 1.2;
v2[1] = 7.89;
v2[2] = 12.34;
```

Compared to **v1**, the “initialization” of **v2** is tedious and error-prone (we deliberately got the number of elements wrong in that code fragment). Using **push\_back()** can save us from mentioning the size:

```
vector v3; // tedious and repetitive
v2.push_back(1.2);
v2.push_back(7.89);
v2.push_back(12.34);
```

But this is still repetitive, so how do we write a constructor that accepts an initializer list as its argument? A **{ }**-delimited list of elements of type **T** is presented to the programmer as an object of the standard library type **initializer\_list<T>**, a list of **T**s, so we can write

```
class vector {
    int sz; // the size
    double* elem; // a pointer to the elements
public:
    vector(int s) // constructor (s is the element count)
        :sz{s}, elem{new double[sz]} // uninitialized memory for elements
    {
        for (int i = 0; i < sz; ++i) elem[i] = 0.0; // initialize
    }

    vector(initializer_list<double> lst) // initializer-list constructor
        :sz{lst.size()}, elem{new double[sz]} // uninitialized memory
        // for elements
    {
        copy(lst.begin(), lst.end(), elem); // initialize (using std::copy(); §B.5.2)
    }
    // ...
};
```

We used the standard library **copy** algorithm (§B.5.2). It copies a sequence of elements specified by its first two arguments (here, the beginning and the end of the **initializer\_list**) to a sequence of elements starting with its third argument (here, the **vector**'s elements starting at **elem**).

Now we can write

```
vector v1 = {1,2,3}; // three elements 1.0, 2.0, 3.0
vector v2(3); // three elements each with the (default) value 0.0
```

Note how we use `()` for an element count and `{ }` for element lists. We need a notation to distinguish them. For example:

```
vector v1 {3};           // one element with the value 3.0
vector v2(3);           // three elements each with the (default) value 0.0
```

This is not very elegant, but it is effective. If there is a choice, the compiler will interpret a value in a `{ }` list as an element value and pass it to the initializer-list constructor as an element of an **initializer\_list**.

In most cases – including all cases we will encounter in this book – the `=` before an `{ }` initializer list is optional, so we can write

```
vector v11 = {1,2,3};    // three elements 1.0, 2.0, 3.0
vector v12 {1,2,3};    // three elements 1.0, 2.0, 3.0
```

The difference is purely one of style.

Note that we pass **initializer\_list<double>** by value. That was deliberate and required by the language rules: an **initializer\_list** is simply a handle to elements allocated “elsewhere” (see §B.6.4).

## 18.3 Copying

Consider again our incomplete **vector**:

```
class vector {
    int sz;           // the size
    double* elem;     // a pointer to the elements
public:
    vector(int s)          // constructor
        :sz{s}, elem{new double[s]} { /* ... */ }
    ~vector()          // destructor
        { delete[] elem; }      // deallocates memory
    // ...
};
```

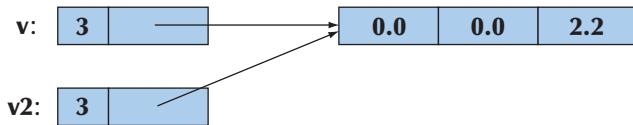
Let’s try to copy one of these vectors:

```
void f(int n)
{
    vector v(3);           // define a vector of 3 elements
    v.set(2,2.2);         // set v[2] to 2.2
```

```
vector v2 = v;      // what happens here?
// ...
}
```

Ideally, `v2` becomes a copy of `v` (that is, `=` makes copies); that is, `v2.size() == v.size()` and `v2[i] == v[i]` for all `i` in the range `[0:v.size())`. Furthermore, all memory is returned to the free store upon exit from `f()`. That's what the standard library `vector` does (of course), but it's not what happens for our still-far-too-simple `vector`. Our task is to improve our `vector` to get it to handle such examples correctly, but first let's figure out what our current version actually does. Exactly what does it do wrong? How? And why? Once we know that, we can probably fix the problems. More importantly, we have a chance to recognize and avoid similar problems when we see them in other contexts.

The default meaning of copying for a class is “Copy all the data members.” That often makes perfect sense. For example, we copy a `Point` by copying its coordinates. But for a pointer member, just copying the members causes problems. In particular, for the `vectors` in our example, it means that after the copy, we have `v.sz == v2.sz` and `v.elem == v2.elem` so that our `vectors` look like this:



That is, `v2` doesn't have a copy of `v`'s elements; it shares `v`'s elements. We could write

```
v.set(1,99);      // set v[1] to 99
v2.set(0,88);     // set v2[0] to 88
cout << v.get(0) << ' ' << v2.get(1);
```

The result would be the output **88 99**. That wasn't what we wanted. Had there been no “hidden” connection between `v` and `v2`, we would have gotten the output **0 0**, because we never wrote to `v[0]` or to `v2[1]`. You could argue that the behavior we got is “interesting,” “neat!” or “sometimes useful,” but that is not what we intended or what the standard library `vector` provides. Also, what happens when we return from `f()` is an unmitigated disaster. Then, the destructors for `v` and `v2` are implicitly called; `v`'s destructor frees the storage used for the elements using

```
delete[] elem;
```

and so does `v2`'s destructor. Since `elem` points to the same memory location in both `v` and `v2`, that memory will be freed twice with likely disastrous results (§17.4.6).

### 18.3.1 Copy constructors

So, what do we do? We'll do the obvious: provide a copy operation that copies the elements and make sure that this copy operation gets called when we initialize one **vector** with another.

Initialization of objects of a class is done by a constructor. So, we need a constructor that copies. Unsurprisingly, such a constructor is called a *copy constructor*. It is defined to take as its argument a reference to the object from which to copy. So, for class **vector** we need

```
vector(const vector&);
```

This constructor will be called when we try to initialize one **vector** with another. We pass by reference because we (obviously) don't want to copy the argument of the constructor that defines copying. We pass by **const** reference because we don't want to modify our argument (§8.5.6). So we refine **vector** like this:

```
class vector {
    int sz;
    double* elem;
public:
    vector(const vector&);           // copy constructor: define copy
    // ...
};
```

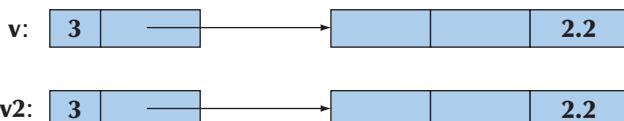
The copy constructor sets the number of elements (**sz**) and allocates memory for the elements (initializing **elem**) before copying element values from the argument **vector**:

```
vector:: vector(const vector& arg)
// allocate elements, then initialize them by copying
    :sz{arg.sz}, elem{new double[arg.sz]}
{
    copy(arg,arg+sz,elem);      // std::copy(); see §B.5.2
}
```

Given this copy constructor, consider again our example:

```
vector v2 = v;
```

This definition will initialize **v2** by a call of **vector**'s copy constructor with **v** as its argument. Again given a **vector** with three elements, we now get



Given that, the destructor can do the right thing. Each set of elements is correctly freed. Obviously, the two **vectors** are now independent so that we can change the value of elements in **v** without affecting **v2** and vice versa. For example:

```
v.set(1,99);           // set v[1] to 99
v2.set(0,88);          // set v2[0] to 88
cout << v.get(0) << ' ' << v2.get(1);
```

This will output **0 0**.

Instead of saying

```
vector v2 = v;
```

we could equally well have said

```
vector v2 {v};
```

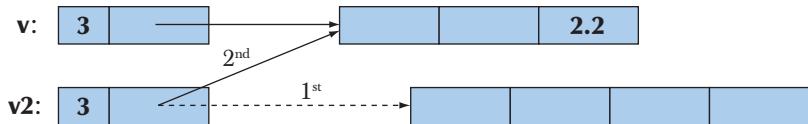
When **v** (the initializer) and **v2** (the variable being initialized) are of the same type and that type has copying conventionally defined, those two notations mean exactly the same thing and you can use whichever notation you like better.

### 18.3.2 Copy assignments

We handle copy construction (initialization), but we can also copy **vectors** by assignment. As with copy initialization, the default meaning of copy assignment is memberwise copy, so with **vector** as defined so far, assignment will cause a double deletion (exactly as shown for copy constructors in §18.3.1) plus a memory leak. For example:

```
void f2(int n)
{
    vector v(3);           // define a vector
    v.set(2,2.2);
    vector v2(4);
    v2 = v;              // assignment: what happens here?
    // ...
}
```

We would like **v2** to be a copy of **v** (and that's what the standard library **vector** does), but since we have said nothing about the meaning of assignment of our **vector**, the default assignment is used; that is, the assignment is a memberwise copy so that **v2**'s **sz** and **elem** become identical to **v**'s **sz** and **elem**, respectively. We can illustrate that like this:



When we leave **f2()**, we have the same disaster as we had when leaving **f()** in §18.3 before we added the copy constructor: the elements pointed to by both **v** and **v2** are freed twice (using **delete[]**). In addition, we have leaked the memory initially allocated for **v2**'s four elements. We “forgot” to free those. The remedy for this copy assignment is fundamentally the same as for the copy initialization (§18.3.1). We define an assignment that copies properly:

```

class vector {
    int sz;
    double* elem;
public:
    vector& operator=(const vector&);      // copy assignment
    // . . .
};

vector& vector::operator=(const vector& a)
    // make this vector a copy of a
{
    double* p = new double[a.sz];           // allocate new space
    copy(a.elem,a.elem+a.sz,elem);          // copy elements
    delete[] elem;                         // deallocate old space
    elem = p;                            // now we can reset elem
    sz = a.sz;
    return *this;                         // return a self-reference (see §17.10)
}

```

Assignment is a bit more complicated than construction because we must deal with the old elements. Our basic strategy is to make a copy of the elements from the source **vector**:

```

double* p = new double[a.sz];           // allocate new space
copy(a.elem,a.elem+a.sz,elem);          // copy elements

```

Then we free the old elements from the target **vector**:

```

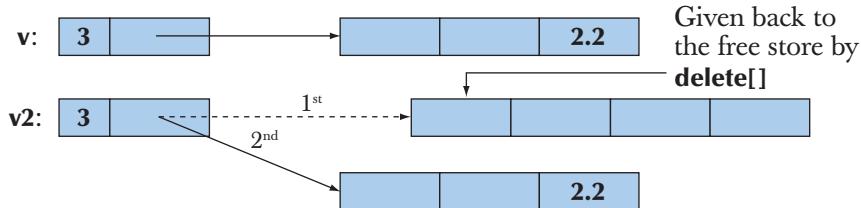
delete[] elem;                         // deallocate old space

```

Finally, we let **elem** point to the new elements:

```
elem = p;           // now we can reset elem
sz = a.sz;
```

We can represent the result graphically like this:



We now have a **vector** that doesn't leak memory and doesn't free (**delete[]**) any memory twice.

When implementing the assignment, you could consider simplifying the code by freeing the memory for the old elements before creating the copy, but it is usually a very good idea not to throw away information before you know that you can replace it. Also, if you did that, strange things would happen if you assigned a **vector** to itself:

```
vector v(10);
v = v;    // self-assignment
```

Please check that our implementation handles that case correctly (if not with optimal efficiency).

### 18.3.3 Copy terminology

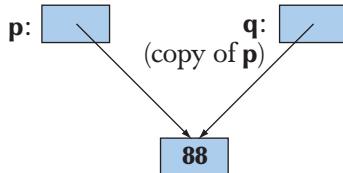
Copying is an issue in most programs and in most programming languages. The basic issue is whether you copy a pointer (or reference) or copy the information pointed to (referred to):

- *Shallow copy* copies only a pointer so that the two pointers now refer to the same object. That's what pointers and references do.
- *Deep copy* copies what a pointer points to so that the two pointers now refer to distinct objects. That's what **vectors**, **strings**, etc. do. We define copy constructors and copy assignments when we want deep copy for objects of our classes.

Here is an example of shallow copy:

```
int* p = new int{77};
int* q = p;           // copy the pointer p
*p = 88;             // change the value of the int pointed to by p and q
```

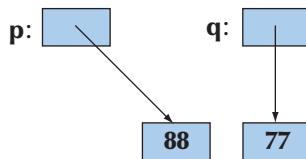
We can illustrate that like this:



In contrast, we can do a deep copy:

```
int* p = new int{77};
int* q = new int{*p}; // allocate a new int, then copy the value pointed to by p
*p = 88;             // change the value of the int pointed to by p
```

We can illustrate that like this:



Using this terminology, we can say that the problem with our original **vector** was that it did a shallow copy, rather than copying the elements pointed to by its **elem** pointer. Our improved **vector**, like the standard library **vector**, does a deep copy by allocating new space for the elements and copying their values. Types that provide shallow copy (like pointers and references) are said to have *pointer semantics* or *reference semantics* (they copy addresses). Types that provide deep copy (like **string** and **vector**) are said to have *value semantics* (they copy the values pointed to). From a user perspective, types with value semantics behave as if no pointers were involved – just values that can be copied. One way of thinking of types with value semantics is that they “work just like integers” as far as copying is concerned.



### 18.3.4 Moving

If a **vector** has a lot of elements, it can be expensive to copy. So, we should copy **vectors** only when we need to. Consider an example:

```
vector fill(istream& is)
{
```

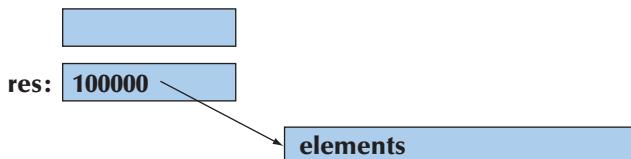
```

vector res;
for (double x; is>>x; ) res.push_back(x);
return res;
}

void use()
{
    vector vec = fill(cin);
    // ... use vec ...
}

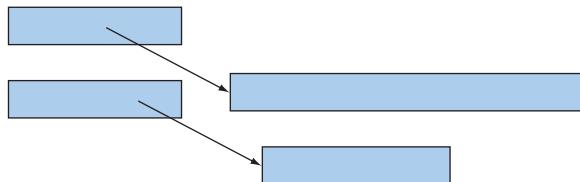
```

Here, we fill the local vector `res` from the input stream and return it to `use()`. Copying `res` out of `fill()` and into `vec` could be expensive. But why copy? We don't want a copy! We can never use the original (`res`) after the return. In fact, `res` is destroyed as part of the return from `fill()`. So how can we avoid the copy? Consider again how a vector is represented in memory:



We would like to “steal” the representation of `res` to use for `vec`. In other words, we would like `vec` to refer to the elements of `res` without any copy.

After moving `res`'s element pointer and element count to `vec`, `res` holds no elements. We have successfully moved the value from `res` out of `fill()` to `vec`. Now, `res` can be destroyed (simply and efficiently) without any undesirable side effects:



We have successfully moved 100,000 `doubles` out of `fill()` and into its caller at the cost of four single-word assignments.

How do we express such a move in C++ code? We define move operations to complement the copy operations:

```

class vector {
    int sz;
    double* elem;

```

```
public:
    vector(vector&& a);           // move constructor
    vector& operator=(vector&&);   // move assignment
    // ...
};
```

The funny `&&` notation is called an “rvalue reference.” We use it for defining move operations. Note that move operations do not take `const` arguments; that is, we write `(vector&&)` and not `(const vector&&)`. Part of the purpose of a move operation is to modify the source, to make it “empty.” The definitions of move operations tend to be simple. They tend to be simpler and more efficient than their copy equivalents. For `vector`, we get

```
vector::vector(vector&& a)
    :sz{a.sz}, elem{a.elem}      // copy a's elem and sz
{
    a.sz = 0;                  // make a the empty vector
    a.elem = nullptr;
}

vector& vector::operator=(vector&& a) // move a to this vector
{
    delete[] elem;            // deallocate old space
    elem = a.elem;            // copy a's elem and sz
    sz = a.sz;
    a.elem = nullptr;          // make a the empty vector
    a.sz = 0;
    return *this;           // return a self-reference (see §17.10)
}
```

By defining a move constructor, we make it easy and cheap to move around large amounts of information, such as a vector with many elements. Consider again:

```
vector fill(istream& is)
{
    vector res;
    for (double x; is>>x; ) res.push_back(x);
    return res;
}
```

The move constructor is implicitly used to implement the return. The compiler knows that the local value returned (`res`) is about to go out of scope, so it can move from it, rather than copying.

The importance of move constructors is that we do not have to deal with pointers or references to get large amounts of information out of a function. Consider this flawed (but conventional) alternative:

```
vector* fill2(istream& is)
{
    vector* res = new vector;
    for (double x; is>>x; ) res->push_back(x);
    return res;
}

void use2()
{
    vector* vec = fill(cin);
    // ... use vec ...
    delete vec;
}
```

Now we have to remember to delete the `vector`. As described in §17.4.6, deleting objects placed on the free store is not as easy to do consistently and correctly as it might seem.

## 18.4 Essential operations

We have now reached the point where we can discuss how to decide which constructors a class should have, whether it should have a destructor, and whether you need to provide copy and move operations. There are seven essential operations to consider:

- Constructors from one or more arguments
- Default constructor
- Copy constructor (copy object of same type)
- Copy assignment (copy object of same type)
- Move constructor (move object of same type)
- Move assignment (move object of same type)
- Destructor

Usually we need one or more constructors that take arguments needed to initialize an object. For example:

```
string s {"cat.jpg"}; // initialize s to the character string "cat.jpg"
Image ii {Point{200,300}, "cat.jpg"}; // initialize a Point with the
// coordinates{200,300},
// then display the contents of file
// cat.jpg at that Point
```

The meaning/use of an initializer is completely up to the constructor. The standard **string**'s constructor uses a character string as an initial value, whereas **Image**'s constructor uses the string as the name of a file to open. Usually we use a constructor to establish an invariant (§9.4.3). If we can't define a good invariant for a class that its constructors can establish, we probably have a poorly designed class or a plain data structure.

Constructors that take arguments are as varied as the classes they serve. The remaining operations have more regular patterns.

How do we know if a class needs a default constructor? We need a default constructor if we want to be able to make objects of the class without specifying an initializer. The most common example is when we want to put objects of a class into a standard library **vector**. The following works only because we have default values for **int**, **string**, and **vector<int>**:

```
vector<double> vi(10); // vector of 10 doubles, each initialized to 0.0
vector<string> vs(10); // vector of 10 strings, each initialized to ""
vector<vector<int>> vvi(10); // vector of 10 vectors, each initialized to vector{}
```

So, having a default constructor is often useful. The question then becomes: “When does it make sense to have a default constructor?” An answer is: “When we can establish the invariant for the class with a meaningful and obvious default value.” For value types, such as **int** and **double**, the obvious value is **0** (for **double**, that becomes **0.0**). For **string**, the empty **string**, **""**, is the obvious choice. For **vector**, the empty **vector** serves well. For every type **T**, **T{}** is the default value, if a default exists. For example, **double{}** is **0.0**, **string{}** is **""**, and **vector<int>{}** is the empty **vector** of **ints**.

A class needs a destructor if it acquires resources. A resource is something you “get from somewhere” and that you must give back once you have finished using it. The obvious example is memory that you get from the free store (using **new**) and have to give back to the free store (using **delete** or **delete[]**). Our **vector** acquires memory to hold its elements, so it has to give that memory back; therefore, it needs a destructor. Other resources that you might encounter as your programs increase in ambition and sophistication are files (if you open one, you also have to close it), locks, thread handles, and sockets (for communication with processes and remote computers).

Another sign that a class needs a destructor is simply that it has members that are pointers or references. If a class has a pointer or a reference member, it often needs a destructor and copy operations.

A class that needs a destructor almost always also needs a copy constructor and a copy assignment. The reason is simply that if an object has acquired a resource (and has a pointer member pointing to it), the default meaning of copy (shallow, memberwise copy) is almost certainly wrong. Again, **vector** is the classic example.

Similarly, a class that needs a destructor almost always also needs a move constructor and a move assignment. The reason is simply that if an object has acquired a resource (and has a pointer member pointing to it), the default meaning of copy (shallow, memberwise copy) is almost certainly wrong and the usual remedy (copy operations that duplicate the complete object state) can be expensive. Again, **vector** is the classic example.

In addition, a base class for which a derived class may have a destructor needs a **virtual** destructor (§17.5.2).

### 18.4.1 Explicit constructors

A constructor that takes a single argument defines a conversion from its argument type to its class. This can be most useful. For example:

```
class complex {  
public:  
    complex(double);      // defines double-to-complex conversion  
    complex(double,double);  
    // ...  
};  
  
complex z1 = 3.14;           // OK: convert 3.14 to (3.14,0)  
complex z2 = complex{1.2, 3.4};
```

However, implicit conversions should be used sparingly and with caution, because they can cause unexpected and undesirable effects. For example, our **vector**, as defined so far, has a constructor that takes an **int**. This implies that it defines a conversion from **int** to **vector**. For example:

```
class vector {  
    // ...  
    vector(int);  
    // ...  
};
```

```

vector v = 10;           // odd: makes a vector of 10 doubles
v = 20;                 // eh? Assigns a new vector of 20 doubles to v

void f(const vector&);
f(10);                  // eh? Calls f with a new vector of 10 doubles

```

It seems we are getting more than we have bargained for. Fortunately, it is simple to suppress this use of a constructor as an implicit conversion. A constructor-defined **explicit** provides only the usual construction semantics and not the implicit conversions. For example:

```

class vector {
    // ...
    explicit vector(int);
    // ...
};

vector v = 10;           // error: no int-to-vector conversion
v = 20;                 // error: no int-to-vector conversion
vector v0(10);          // OK

void f(const vector&);
f(10);                  // error: no int-to-vector<double> conversion
f(vector(10));          // OK

```

To avoid surprising conversions, we – and the standard – define **vector**'s single-argument constructors to be **explicit**. It's a pity that constructors are not **explicit** by default; if in doubt, make any constructor that can be invoked with a single argument **explicit**.

### 18.4.2 Debugging constructors and destructors

Constructors and destructors are invoked at well-defined and predictable points of a program's execution. However, we don't always write **explicit** calls, such as **vector(2)**; rather we do something, such as declaring a **vector**, passing a **vector** as a by-value argument, or creating a **vector** on the free store using **new**. This can cause confusion for people who think in terms of syntax. There is not just a single syntax that triggers a constructor. It is simpler to think of constructors and destructors this way:

- Whenever an object of type **X** is created, one of **X**'s constructors is invoked.
- Whenever an object of type **X** is destroyed, **X**'s destructor is invoked.

A destructor is called whenever an object of its class is destroyed; that happens when names go out of scope, the program terminates, or **delete** is used on a pointer to an object. A constructor (some appropriate constructor) is invoked whenever an object of its class is created; that happens when a variable is initialized, when an object is created using **new** (except for built-in types), and whenever an object is copied.

But when does that happen? A good way to get a feel for that is to add print statements to constructors, assignment operations, and destructors and then just try. For example:

```
struct X {          // simple test class
    int val;

    void out(const string& s, int nv)
        { cerr << this << " ->" << s << ":" << val << "(" << nv << ")\\n"; }

    X(){ out("X()",0); val=0; }           // default constructor
    X(int v) { val=v; out( "X(int)",v); }
    X(const X& x){ val=x.val; out("X(X& ) ",x.val); } // copy constructor
    X& operator=(const X& a)           // copy assignment
        { out("X::operator=()",a.val); val=a.val; return *this; }
    ~X() { out("~X()",0); }           // destructor
};
```

Anything we do with this **X** will leave a trace that we can study. For example:

```
X glob(2);          // a global variable

X copy(X a) { return a; }

X copy2(X a) { X aa = a; return aa; }

X& ref_to(X& a) { return a; }

X* make(int i) { X a(i); return new X(a); }

struct XX { X a; X b; };

int main()
{
    X loc {4};      // local variable
    X loc2 {loc};   // copy construction
```

```
loc = X{5};           // copy assignment
loc2 = copy(loc);    // call by value and return
loc2 = copy2(loc);
X loc3 {6};
X& r = ref_to(loc); // call by reference and return
delete make(7);
delete make(8);
vector<X> v(4);    // default values
XX loc4;
X* p = new X{9};    // an X on the free store
delete p;
X* pp = new X[5];   // an array of Xs on the free store
delete[] pp;
}
```

Try executing that.

---

### TRY THIS



We really mean it: do run this example and make sure you understand the result. If you do, you'll understand most of what there is to know about construction and destruction of objects.

Depending on the quality of your compiler, you may note some “missing copies” relating to our calls of `copy()` and `copy2()`. We (humans) can see that those functions do nothing: they just copy a value unmodified from input to output. If a compiler is smart enough to notice that, it is allowed to eliminate the calls to the copy constructor. In other words, a compiler is allowed to assume that a copy constructor copies and does nothing but copy. Some compilers are smart enough to eliminate many spurious copies. However, compilers are not guaranteed to be that smart, so if you want portable performance, consider move operations (§18.3.4).

Now consider: Why should we bother with this “silly class `X`”? It’s a bit like the finger exercises that musicians have to do. After doing them, other things – things that matter – become easier. Also, if you have problems with constructors and destructors, you can insert such print statements in constructors for your real classes to see that they work as intended. For larger programs, this exact kind of tracing becomes tedious, but similar techniques apply. For example, you can determine whether you have a memory leak by seeing if the number of constructions minus the number of destructions equals zero. Forgetting to define copy constructors and copy assignments for classes that allocate memory or hold pointers to objects is a common – and easily avoidable – source of problems.





If your problems get too big to handle by such simple means, you will have learned enough to be able to start using the professional tools for finding such problems; they are often referred to as “leak detectors.” The ideal, of course, is not to leak memory by using techniques that avoid such leaks.

## 18.5 Access to vector elements

So far (§17.6), we have used `set()` and `get()` member functions to access elements. Such uses are verbose and ugly. We want our usual subscript notation: `v[i]`. The way to get that is to define a member function called `operator[]`. Here is our first (naive) try:

```
class vector {
    int sz;           // the size
    double* elem;    // a pointer to the elements
public:
    ...
    double operator[](int n) { return elem[n]; }    // return element
};
```

That looks good and especially it looks simple, but unfortunately it is too simple. Letting the subscript operator (`operator[](0)`) return a value enables reading but not writing of elements:

```
vector v(10);
double x = v[2];          // fine
v[3] = x;                 // error: v[3] is not an lvalue
```

Here, `v[i]` is interpreted as a call `v.operator[](i)`, and that call returns the value of `v`'s element number `i`. For this overly naive `vector`, `v[3]` is a floating-point value, not a floating-point variable.

---

### TRY THIS



Make a version of this `vector` that is complete enough to compile and see what error message your compiler produces for `v[3]=x;`.

Our next try is to let `operator[]` return a pointer to the appropriate element:

```
class vector {
    int sz;           // the size
    double* elem;    // a pointer to the elements
```

```
public:
    ...
    double* operator[](int n) { return &elem[n]; }      // return pointer
};
```

Given that definition, we can write

```
vector v(10);
for (int i=0; i<v.size(); ++i) {      // works, but still too ugly
    *v[i] = i;
    cout << *v[i];
}
```

Here, **v[i]** is interpreted as a call **v.operator[](i)**, and that call returns a pointer to **v**'s element number **i**. The problem is that we have to write **\*** to dereference that pointer to get to the element. That's almost as bad as having to write **set()** and **get()**. Returning a reference from the subscript operator solves this problem:

```
class vector {
    ...
    double& operator[ ](int n) { return elem[n]; }  // return reference
};
```

Now we can write

```
vector v(10);
for (int i=0; i<v.size(); ++i) {      // works!
    v[i] = i;                      // v[i] returns a reference element i
    cout << v[i];
}
```

We have achieved the conventional notation: **v[i]** is interpreted as a call **v.operator[](i)**, and that returns a reference to **v**'s element number **i**.

### 18.5.1 Overloading on **const**

The **operator[]()** defined so far has a problem: it cannot be invoked for a **const vector**. For example:

```
void f(const vector& cv)
{
    double d = cv[1];          // error, but should be fine
    cv[1] = 2.0;               // error (as it should be)
}
```

The reason is that our `vector::operator[](0)` could potentially change a `vector`. It doesn't, but the compiler doesn't know that because we "forgot" to tell it. The solution is to provide a version that is a `const` member function (see §9.7.4). That's easily done:

```
class vector {
    // ...
    double& operator[](int n);           // for non-const vectors
    double operator[](int n) const;      // for const vectors
};
```

We obviously couldn't return a `double&` from the `const` version, so we returned a `double` value. We could equally well have returned a `const double&`, but since a `double` is a small object there would be no point in returning a reference (§8.5.6), so we decided to pass it back by value. We can now write

```
void ff(const vector& cv, vector& v)
{
    double d = cv[1];                  // fine (uses the const [])
    cv[1] = 2.0;                      // error (uses the const [])
    double d = v[1];                  // fine (uses the non-const [])
    v[1] = 2.0;                       // fine (uses the non-const [])
}
```

Since `vectors` are often passed by `const` reference, this `const` version of `operator[](0)` is an essential addition.

## 18.6 Arrays

For a while, we have used *array* to refer to a sequence of objects allocated on the free store. We can also allocate arrays elsewhere as named variables. In fact, they are common

- As global variables (but global variables are most often a bad idea)
- As local variables (but arrays have serious limitations there)
- As function arguments (but an array doesn't know its own size)
- As class members (but member arrays can be hard to initialize)

Now, you might have detected that we have a not-so-subtle bias in favor of `vectors` over arrays. Use `std::vector` where you have a choice – and you have a choice in most contexts. However, arrays existed long before `vectors` and are roughly equivalent to what is offered in other languages (notably C), so you must know

arrays, and know them well, to be able to cope with older code and with code written by people who don't appreciate the advantages of **vector**.

So, what is an array? How do we define an array? How do we use an array? An *array* is a homogeneous sequence of objects allocated in contiguous memory; that is, all elements of an array have the same type and there are no gaps between the objects of the sequence. The elements of an array are numbered from 0 upward. In a declaration, an array is indicated by “square brackets”:

```
const int max = 100;
int gai[max];           // a global array (of 100 ints); "lives forever"

void f(int n)
{
    char lac[20];      // local array; "lives" until the end of scope
    int lai[60];
    double lad[n];    // error: array size not a constant
    // ...
}
```

Note the limitation: the number of elements of a named array must be known at compile time. If you want the number of elements to be a variable, you must put it on the free store and access it through a pointer. That's what **vector** does with its array of elements.

Just like the arrays on the free store, we access named arrays using the subscript and dereference operators (`[]` and `*`). For example:

```
void f2()
{
    char lac[20];      // local array; "lives" until the end of scope

    lac[7] = 'a';
    *lac = 'b';        // equivalent to lac[0]='b'

    lac[-2] = 'b';     // huh?
    lac[200] = 'c';    // huh?
}
```

This function compiles, but we know that “compiles” doesn't mean “works correctly.” The use of `[]` is obvious, but there is no range checking, so `f2()` compiles, and the result of writing to `lac[-2]` and `lac[200]` is (as for all out-of-range access) usually disastrous. Don't do it. Arrays do not range check. Again, we are dealing directly with physical memory here; don't expect “system support.”



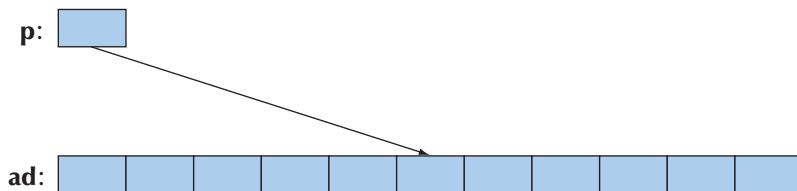
But couldn't the compiler see that `lac` has just 20 elements so that `lac[200]` is an error? A compiler could, but as far as we know no production compiler does. The problem is that keeping track of array bounds at compile time is impossible in general, and catching errors in the simplest cases (like the one above) only is not very helpful.

### 18.6.1 Pointers to array elements

A pointer can point to an element of an array. Consider:

```
double ad[10];
double* p = &ad[5];           // point to ad[5]
```

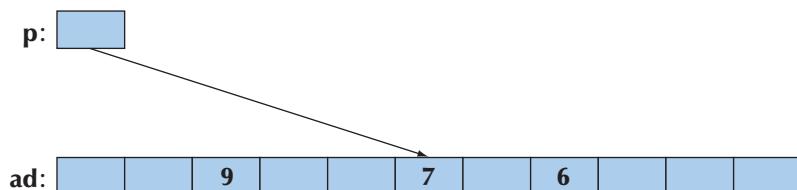
We now have a pointer `p` to the `double` known as `ad[5]`:



We can subscript and dereference that pointer:

```
*p = 7;
p[2] = 6;
p[-3] = 9;
```

We get

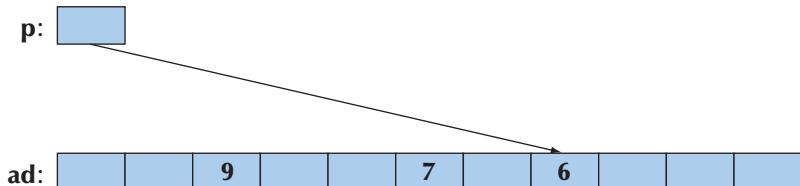


That is, we can subscript the pointer with both positive and negative numbers. As long as the resulting element is in range, all is well. However, access outside the range of the array pointed into is illegal (as with free-store-allocated arrays; see §17.4.3). Typically, access outside an array is not detected by the compiler and (sooner or later) is disastrous.

Once a pointer points into an array, addition and subscripting can be used to make it point to another element of the array. For example:

```
p += 2;           // move p 2 elements to the right
```

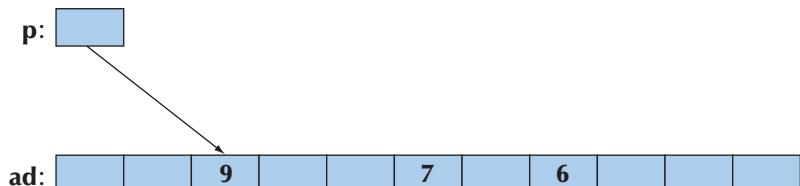
We get



And

```
p -= 5;           // move p 5 elements to the left
```

We get



Using **+**, **-**, **+=**, and **-=** to move pointers around is called *pointer arithmetic*. Obviously, if we do that, we have to take great care to ensure that the result is not a pointer to memory outside the array:

```
p += 1000;       // insane: p points into an array with just 10 elements
double d = *p;   // illegal: probably a bad value
                    // (definitely an unpredictable value)
*p = 12.34;     // illegal: probably scrambles some unknown data
```

Unfortunately, not all bad bugs involving pointer arithmetic are that easy to spot. The best policy is usually simply to avoid pointer arithmetic.

The most common use of pointer arithmetic is incrementing a pointer (using **++**) to point to the next element and decrementing a pointer (using **--**) to point

to the previous element. For example, we could print the value of `ad`'s elements like this:

```
for (double* p = &ad[0]; p<&ad[10]; ++p) cout << *p << '\n';
```

Or backward:

```
for (double* p = &ad[9]; p>=&ad[0]; --p) cout << *p << '\n';
```

This use of pointer arithmetic is not uncommon. However, we find the last (“backward”) example quite easy to get wrong. Why `&ad[9]` and not `&ad[10]`? Why `>=` and not `>`? These examples could equally well (and equally efficiently) be done using subscripting. Such examples could be done equally well using subscripting into a `vector`, which is more easily range checked.

Note that most real-world uses of pointer arithmetic involve a pointer passed as a function argument. In that case, the compiler doesn't have a clue how many elements are in the array pointed into: you are on your own. That is a situation we prefer to stay away from whenever we can.

Why does C++ have (allow) pointer arithmetic at all? It can be such a bother and doesn't provide anything new once we have subscripting. For example:

```
double* p1 = &ad[0];
double* p2 = p1+7;
double* p3 = &p1[7];
if (p2 != p3) cout << "impossible!\n";
```

Mainly, the reason is historical. These rules were crafted for C decades ago and can't be removed without breaking a lot of code. Partly, there can be some convenience gained by using pointer arithmetic in some important low-level applications, such as memory managers.

## 18.6.2 Pointers and arrays

The name of an array refers to all the elements of the array. Consider:

```
char ch[100];
```

The size of `ch`, `sizeof(ch)`, is 100. However, the name of an array turns into (“decays to”) a pointer with the slightest excuse. For example:

```
char* p = ch;
```

Here `p` is initialized to `&ch[0]` and `sizeof(p)` is something like 4 (not 100).

This can be useful. For example, consider a function `strlen()` that counts the number of characters in a zero-terminated array of characters:

```
int strlen(const char* p)    // similar to the standard library strlen()
{
    int count = 0;
    while (*p) { ++count; ++p; }
    return count;
}
```

We can now call this with `strlen(ch)` as well as `strlen(&ch[0])`. You might point out that this is a very minor notational advantage, and we'd have to agree.

One reason for having array names convert to pointers is to avoid accidentally passing large amounts of data by value. Consider:

```
int strlen(const char a[])    // similar to the standard library strlen()
{
    int count = 0;
    while (a[count]) { ++count; }
    return count;
}

char lots [100000];

void f()
{
    int nchar = strlen(lots);
    // ...
}
```

Naively (and quite reasonably), you might expect this call to copy the 100,000 characters specified as the argument to `strlen()`, but that's not what happens. Instead, the argument declaration `char p[]` is considered equivalent to `char* p`, and the call `strlen(lots)` is considered equivalent to `strlen(&lots[0])`. This saves you from an expensive copy operation, but it should surprise you. Why should it surprise you? Because in every other case, when you pass an object and don't explicitly declare an argument to be passed by reference (§8.5.3–6), that object is copied.

Note that the pointer you get from treating the name of an array as a pointer to its first element is a value and not a variable, so you cannot assign to it:

```
char ac[10];
ac = new char [20];      // error: no assignment to array name
&ac[0] = new char [20];  // error: no assignment to pointer value
```

Finally! A problem that the compiler will catch!

As a consequence of this implicit array-name-to-pointer conversion, you can't even copy arrays using assignment:

```
int x[100];
int y[100];
// ...
x = y;           // error
int z[100] = y; // error
```

This is consistent, but often a bother. If you need to copy an array, you must write some more elaborate code to do so. For example:

```
for (int i=0; i<100; ++i) x[i]=y[i];      // copy 100 ints
memcpy(x,y,100*sizeof(int));            // copy 100*sizeof(int) bytes
copy(y,y+100, x);                     // copy 100 ints
```

Note that the C language doesn't support anything like **vector**, so in C, you must use arrays extensively. This implies that a lot of C++ code uses arrays (§27.1.2). In particular, C-style strings (zero-terminated arrays of characters; see §27.5) are very common.

If we want assignment, we have to use something like the standard library **vector**. The **vector** equivalent to the copying code above is

```
vector<int> x(100);
vector<int> y(100);
// ...
x = y;           // copy 100 ints
```

### 18.6.3 Array initialization

An array of **chars** can be initialized with a string literal. For example:

```
char ac[] = "Beorn";           // array of 6 chars
```

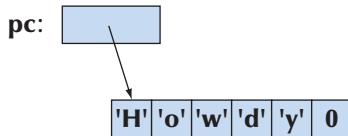
Count those characters. There are five, but **ac** becomes an array of six characters because the compiler adds a terminating zero character at the end of a string literal:

ac:	'B'	'e'	'o'	'r'	'n'	0
-----	-----	-----	-----	-----	-----	---

A zero-terminated string is the norm in C and many systems. We call such a zero-terminated array of characters a *C-style string*. All string literals are C-style strings. For example:

```
char* pc = "Howdy";           // pc points to an array of 6 chars
```

Graphically:



Note that the **char** with the numeric value **0** is not the character '**0**' or any other letter or digit. The purpose of that terminating zero is to allow functions to find the end of the string. Remember: An array does not know its size. Relying on the terminating zero convention, we can write

```
int strlen(const char* p)           // similar to the standard library strlen()
{
    int n = 0;
    while (p[n]) ++n;
    return n;
}
```

Actually, we don't have to define **strlen()** because it is a standard library function defined in the **<string.h>** header (§27.5, §B.11.3). Note that **strlen()** counts the characters, but not the terminating **0**; that is, you need  $n+1$  **chars** to store  $n$  characters in a C-style string.

Only character arrays can be initialized by literal strings, but all arrays can be initialized by a list of values of their element type. For example:

```
int ai[] = { 1, 2, 3, 4, 5, 6 };      // array of 6 ints
int ai2[100] = {0,1,2,3,4,5,6,7,8,9}; // the last 90 elements are initialized to 0
double ad[100] = { };                // all elements initialized to 0.0
char chars[] = {'a', 'b', 'c'};       // no terminating 0!
```

Note that the number of elements of **ai** is six (not seven) and the number of elements for **chars** is three (not four) – the “add a **0** at the end” rule is for literal character strings only. If an array isn't given a size, that size is deduced from the initializer list. That's a rather useful feature. If there are fewer initializer values

than array elements (as in the definitions of `ai2` and `ad`), the remaining elements are initialized by the element type's default value.

#### 18.6.4 Pointer problems

Like arrays, pointers are often overused and misused. Often, the problems people get themselves into involve both pointers and arrays, so we'll summarize the problems here. In particular, all serious problems with pointers involve trying to access something that isn't an object of the expected type, and many of those problems involve access outside the bounds of an array. Here we will consider

- Access through the null pointer
- Access through an uninitialized pointer
- Access off the end of an array
- Access to a deallocated object
- Access to an object that has gone out of scope

In all cases, the practical problem for the programmer is that the actual access looks perfectly innocent; it is “just” that the pointer hasn't been given a value that makes the use valid. Worse (in the case of a write through the pointer), the problem may manifest itself only a long time later when some apparently unrelated object has been corrupted. Let's consider examples:

*Don't access through the null pointer:*



```
int* p = nullptr;
*p = 7;           // ouch!
```

Obviously, in real-world programs, this typically occurs when there is some code in between the initialization and the use. In particular, passing `p` to a function and receiving it as the result from a function are common examples. We prefer not to pass null pointers around, but if you have to, test for the null pointer before use:

```
int* p = fct_that_can_return_a_nullptr();

if (p == nullptr) {
    // do something
}
else {
    // use p
    *p = 7;
}
```

and

```
void fct_that_can_receive_a_nullptr(int* p)
{
    if (p == nullptr) {
        // do something
    }
    else {
        // use p
        *p = 7;
    }
}
```

Using references (§17.9.1) and using exceptions to signal errors (§5.6 and §19.5) are the main tools for avoiding null pointers.

*Do initialize your pointers:*

```
int* p;
*p = 9;           // ouch!
```

In particular, don't forget to initialize pointers that are class members.

*Don't access nonexistent array elements:*

```
int a[10];
int* p = &a[10];
*p = 11;           // ouch!
a[10] = 12;         // ouch!
```

Be careful with the first and last elements of a loop, and try not to pass arrays around as pointers to their first elements. Instead use **vectors**. If you really must use an array in more than one function (passing it as an argument), then be extra careful and pass its size along.

*Don't access through a deleted pointer:*

```
int* p = new int{7};
// ...
delete p;
// ...
*p = 13;           // ouch!
```

The **delete p** or the code after it may have scribbled all over **\*p** or used it for something else. Of all of these problems, we consider this one the hardest to

systematically avoid. The most effective defense against this problem is not to have “naked” **news** that require “naked” **deletes**: use **new** and **delete** in constructors and destructors or use a container, such as **Vector\_ref** (§E.4), to handle **deletes**.

*Don't return a pointer to a local variable:*

```
int* f()
{
    int x = 7;
    // ...
    return &x;
}

// ...

int* p = f();
// ...
*p = 15;           // ouch!
```

The return from **f()** or the code after it may have scribbled all over **\*p** or used it for something else. The reason for that is that the local variables of a function are allocated (on the stack) upon entry to the function and deallocated again at the exit from the function. In particular, destructors are called for local variables of classes with destructors (§17.5.1). Compilers could catch most problems related to returning pointers to local variables, but few do.

Consider a logically equivalent example:

```
vector& ff()
{
    vector x(7);    // 7 elements
    // ...
    return x;
}    // the vector x is destroyed here

// ...

vector& p = ff();
// ...
p[4] = 15;           // ouch!
```

Quite a few compilers catch this variant of the return problem.

It is common for programmers to underestimate these problems. However, many experienced programmers have been defeated by the innumerable varia-

tions and combinations of these simple array and pointer problems. The solution is not to litter your code with pointers, arrays, **news**, and **deletes**. If you do, “being careful” simply isn’t enough in realistically sized programs. Instead, rely on vectors, RAI<sup>I</sup> (“Resource Acquisition Is Initialization”; see §19.5), and other systematic approaches to the management of memory and other resources.



## 18.7 Examples: palindrome

Enough technical examples! Let’s try a little puzzle. A *palindrome* is a word that is spelled the same from both ends. For example, *anna*, *petep*, and *malayalam* are palindromes, whereas *ida* and *homesick* are not. There are two basic ways of determining whether a word is a palindrome:

- Make a copy of the letters in reverse order and compare that copy to the original.
- See if the first letter is the same as the last, then see if the second letter is the same as the second to last, and keep going until you reach the middle.

Here, we’ll take the second approach. There are many ways of expressing this idea in code depending on how we represent the word and how we keep track of how far we have come with the comparison of characters. We’ll write a little program that tests whether words are palindromes in a few different ways just to see how different language features affect the way the code looks and works.

### 18.7.1 Palindromes using string

First, we try a version using the standard library **string** with **int** indices to keep track of how far we have come with our comparison:

```
bool is_palindrome(const string& s)
{
    int first = 0;           // index of first letter
    int last = s.length() - 1; // index of last letter
    while (first < last) {   // we haven't reached the middle
        if (s[first] != s[last]) return false;
        ++first;             // move forward
        --last;               // move backward
    }
    return true;
}
```

We return **true** if we reach the middle without finding a difference. We suggest that you look at this code to convince yourself that it is correct when there are no

letters in the string, just one letter in the string, an even number of letters in the string, and an odd number of letters in the string. Of course, we should not just rely on logic to see that our code is correct. We should also test. We can exercise `is_palindrome()` like this:

```
int main()
{
    for (string s; cin>>s; ) {
        cout << s << " is";
        if (!is_palindrome(s)) cout << " not";
        cout << " a palindrome\n";
    }
}
```

Basically, the reason we are using a `string` is that “`strings` are good for dealing with words.” It is simple to read a whitespace-separated word into a string, and a `string` knows its size. Had we wanted to test `is_palindrome()` with strings containing whitespace, we could have read using `getline()` (§11.5). That would have shown *ah ha* and *as dffd sa* to be palindromes.

### 18.7.2 Palindromes using arrays

What if we didn’t have `strings` (or `vectors`), so that we had to use an array to store the characters? Let’s see:

```
bool is_palindrome(const char s[], int n)
    // s points to the first character of an array of n characters
{
    int first = 0;           // index of first letter
    int last = n-1;          // index of last letter
    while (first < last) {   // we haven't reached the middle
        if (s[first] != s[last]) return false;
        ++first;              // move forward
        --last;                // move backward
    }
    return true;
}
```

To exercise `is_palindrome()`, we first have to get characters read into the array. One way to do that safely (i.e., without risk of overflowing the array) is like this:

```
istream& read_word(istream& is, char* buffer, int max)
    // read at most max-1 characters from is into buffer
```

```

{
    is.width(max);      // read at most max-1 characters in the next >>
    is >> buffer;      // read whitespace-terminated word,
                        // add zero after the last character read into buffer
    return is;
}

```

Setting the `istream`'s width appropriately prevents buffer overflow for the next `>>` operation. Unfortunately, it also means that we don't know if the read terminated by whitespace or by the buffer being full (so that we need to read more characters). Also, who remembers the details of the behavior of `width()` for input? The standard library `string` and `vector` are really better as input buffers because they expand to fit the amount of input. The terminating `\0` character is needed because most popular operations on arrays of characters (C-style strings) assume 0 termination. Using `read_word()` we can write

```

int main()
{
    constexpr int max = 128;
    for (char s[max]; read_word(cin, s, max); ) {
        cout << s << " is";
        if (!is_palindrome(s, strlen(s))) cout << " not";
        cout << " a palindrome\n";
    }
}

```

The `strlen(s)` call returns the number of characters in the array after the call of `read_word()`, and `cout << s` outputs the characters in the array up to the terminating `\0`.

We consider this “array solution” significantly messier than the “`string` solution,” and it gets much worse if we try to seriously deal with the possibility of long strings. See exercise 10.



### 18.7.3 Palindromes using pointers

Instead of using indices to identify characters, we could use pointers:

```

bool is_palindrome(const char* first, const char* last)
    // first points to the first letter, last to the last letter
{
    while (first < last) {          // we haven't reached the middle
        if (*first != *last) return false;
        ++first;                    // move forward
        --last;                     // move backward
    }
}

```

```

    }
    return true;
}

```



Note that we can actually increment and decrement pointers. Increment makes a pointer point to the next element of an array and decrement makes a pointer point to the previous element. If the array doesn't have such a next element or previous element, you have a serious uncaught out-of-range error. That's another problem with pointers.

We call this **is\_palindrome()** like this:

```

int main()
{
    const int max = 128;
    for (char s[max]; read_word(cin,s,max); ) {
        cout << s << " is";
        if (!is_palindrome(&s[0],&s[strlen(s)-1])) cout << " not";
        cout << " a palindrome\n";
    }
}

```

Just for fun, we rewrite **is\_palindrome()** like this:

```

bool is_palindrome(const char* first, const char* last)
    // first points to the first letter, last to the last letter
{
    if (first<last) {
        if (*first!=*last) return false;
        return is_palindrome(first+1,last-1);
    }
    return true;
}

```

This code becomes obvious when we rephrase the definition of *palindrome*: a word is a palindrome if the first and the last characters are the same and if the substring you get by removing the first and the last characters is a palindrome.



## Drill

In this chapter, we have two drills: one to exercise arrays and one to exercise **vectors** in roughly the same manner. Do both and compare the effort involved in each.

### Array drill:

1. Define a global **int** array **ga** of ten **ints** initialized to 1, 2, 4, 8, 16, etc.
2. Define a function **f()** taking an **int** array argument and an **int** argument indicating the number of elements in the array.
3. In **f()**:
  - a. Define a local **int** array **la** of ten **ints**.
  - b. Copy the values from **ga** into **la**.
  - c. Print out the elements of **la**.
  - d. Define a pointer **p** to **int** and initialize it with an array allocated on the free store with the same number of elements as the argument array.
  - e. Copy the values from the argument array into the free-store array.
  - f. Print out the elements of the free-store array.
  - g. Deallocate the free-store array.
4. In **main()**:
  - a. Call **f()** with **ga** as its argument.
  - b. Define an array **aa** with ten elements, and initialize it with the first ten factorial values (1, 2\*1, 3\*2\*1, 4\*3\*2\*1, etc.).
  - c. Call **f()** with **aa** as its argument.

### Standard library **vector** drill:

1. Define a global **vector<int>** **gv**; initialize it with ten **ints**, 1, 2, 4, 8, 16, etc.
2. Define a function **f()** taking a **vector<int>** argument.
3. In **f()**:
  - a. Define a local **vector<int>** **lv** with the same number of elements as the argument **vector**.
  - b. Copy the values from **gv** into **lv**.
  - c. Print out the elements of **lv**.
  - d. Define a local **vector<int>** **lv2**; initialize it to be a copy of the argument **vector**.
  - e. Print out the elements of **lv2**.
4. In **main()**:
  - a. Call **f()** with **gv** as its argument.
  - b. Define a **vector<int>** **vv**, and initialize it with the first ten factorial values (1, 2\*1, 3\*2\*1, 4\*3\*2\*1, etc.).
  - c. Call **f()** with **vv** as its argument.

## Review

1. What does “Caveat emptor!” mean?
2. What is the default meaning of copying for class objects?
3. When is the default meaning of copying of class objects appropriate? When is it inappropriate?
4. What is a copy constructor?
5. What is a copy assignment?
6. What is the difference between copy assignment and copy initialization?
7. What is shallow copy? What is deep copy?
8. How does the copy of a **vector** compare to its source?
9. What are the five “essential operations” for a class?
10. What is an **explicit** constructor? Where would you prefer one over the (default) alternative?
11. What operations may be invoked implicitly for a class object?
12. What is an array?
13. How do you copy an array?
14. How do you initialize an array?
15. When should you prefer a pointer argument over a reference argument? Why?
16. What is a C-style string?
17. What is a palindrome?

## Terms

array	deep copy	move assignment
array initialization	default constructor	move construction
copy assignment	essential operations	palindrome
copy constructor	<b>explicit</b> constructor	shallow copy

## Exercises

1. Write a function, **char\* strdup(const char\*)**, that copies a C-style string into memory it allocates on the free store. Do not use any standard library functions. Do not use subscripting; use the dereference operator **\*** instead.
2. Write a function, **char\* findx(const char\* s, const char\* x)**, that finds the first occurrence of the C-style string **x** in **s**. Do not use any standard library functions. Do not use subscripting; use the dereference operator **\*** instead.
3. Write a function, **int strcmp(const char\* s1, const char\* s2)**, that compares C-style strings. Let it return a negative number if **s1** is lexicographically

before `s2`, zero if `s1` equals `s2`, and a positive number if `s1` is lexicographically after `s2`. Do not use any standard library functions. Do not use subscripting; use the dereference operator `*` instead.

4. Consider what happens if you give `strdup()`, `findx()`, and `strcmp()` an argument that is not a C-style string. Try it! First figure out how to get a `char*` that doesn't point to a zero-terminated array of characters and then use it (never do this in real – non-experimental – code; it can create havoc). Try it with free-store-allocated and stack-allocated “fake C-style strings.” If the results still look reasonable, turn off debug mode. Redesign and re-implement those three functions so that they take another argument giving the maximum number of elements allowed in argument strings. Then, test that with correct C-style strings and “bad” strings.
5. Write a function, `string cat_dot(const string& s1, const string& s2)`, that concatenates two strings with a dot in between. For example, `cat_dot("Niels", "Bohr")` will return a string containing `Niels.Bohr`.
6. Modify `cat_dot()` from the previous exercise to take a string to be used as the separator (rather than dot) as its third argument.
7. Write versions of the `cat_dot()`s from the previous exercises to take C-style strings as arguments and return a free-store-allocated C-style string as the result. Do not use standard library functions or types in the implementation. Test these functions with several strings. Be sure to free (using `delete`) all the memory you allocated from free store (using `new`). Compare the effort involved in this exercise with the effort involved for exercises 5 and 6.
8. Rewrite all the functions in §18.7 to use the approach of making a backward copy of the string and then comparing; for example, take `"home"`, generate `"emoh"`, and compare those two strings to see that they are different, so `home` isn't a palindrome.
9. Consider the memory layout in §17.4. Write a program that tells the order in which static storage, the stack, and the free store are laid out in memory. In which direction does the stack grow: upward toward higher addresses or downward toward lower addresses? In an array on the free store, are elements with higher indices allocated at higher or lower addresses?
10. Look at the “array solution” to the palindrome problem in §18.7.2. Fix it to deal with long strings by (a) reporting if an input string was too long and (b) allowing an arbitrarily long string. Comment on the complexity of the two versions.
11. Look up (e.g., on the web) *skip list* and implement that kind of list. This is not an easy exercise.
12. Implement a version of the game “Hunt the Wumpus.” “Hunt the Wumpus” (or just “Wump”) is a simple (non-graphical) computer game originally invented by Gregory Yob. The basic premise is that a rather smelly

monster lives in a dark cave consisting of connected rooms. Your job is to slay the wumpus using bow and arrow. In addition to the wumpus, the cave has two hazards: bottomless pits and giant bats. If you enter a room with a bottomless pit, it's the end of the game for you. If you enter a room with a bat, the bat picks you up and drops you into another room. If you enter the room with the wumpus or he enters yours, he eats you. When you enter a room you will be told if a hazard is nearby:

“I smell the wumpus”: It’s in an adjoining room.

“I feel a breeze”: One of the adjoining rooms is a bottomless pit.

“I hear a bat”: A giant bat is in an adjoining room.

For your convenience, rooms are numbered. Every room is connected by tunnels to three other rooms. When entering a room, you are told something like “You are in room 12; there are tunnels to rooms 1, 13, and 4; move or shoot?” Possible answers are **m13** (“Move to room 13”) and **s13-4-3** (“Shoot an arrow through rooms 13, 4, and 3”). The range of an arrow is three rooms. At the start of the game, you have five arrows. The snag about shooting is that it wakes up the wumpus and he moves to a room adjoining the one he was in – that could be your room.

Probably the trickiest part of the exercise is to make the cave by selecting which rooms are connected with which other rooms. You’ll probably want to use a random number generator (e.g., **randint()** from **std\_lib\_facilities.h**) to make different runs of the program use different caves and to move around the bats and the wumpus. Hint: Be sure to have a way to produce a debug output of the state of the cave.

## Postscript

The standard library **vector** is built from lower-level memory management facilities, such as pointers and arrays, and its primary role is to help us avoid the complexities of those facilities. Whenever we design a class, we must consider initialization, copying, and destruction.

*This page intentionally left blank*

# Index

- !. See* Not, 1087
- !=. See* Not equal (inequality), 67, 1088, 1101
- "...". See* String literal, 62
- #. See* Preprocessor directives, 1129
- \$. See* End of line, 873, 1178
- %.* *See*
  - Output format specifier, 1187
  - Remainder (modulo), 68
- %=. See* Remainder and assign, 1090
- &. See*
  - Address of, 588, 1087
  - Bitwise logical operations (and), 956, 1089, 1094
  - Reference to (in declarations), 276–279, 1099
- &&. See* Logical and, 1089, 1094
- &=. See* Bitwise logical operations (and and assign), 1090
- ...!. See* Character literals, 161, 1079–1080
- 0. See*
  - Expression (grouping), 95, 867, 873, 876
  - Function call, 285, 766
  - Function of (in declarations), 113–115, 1099
  - Regular expression (grouping), 1178
- \*. See*
  - Contents of (dereference), 594
  - Multiply, 1088
  - Pointer to (in declarations), 587, 1099
  - Repetition (in **regex**), 868, 873–874, 1178
- \*/ end of block comment, 238*
- \*=. See* Multiply and assign (scale), 67
- +. See*
  - Add, 66, 1088
  - Concatenation (of **strings**), 68–69, 851, 1176
  - Repetition in **regex**, 873–875, 1178
- ++.* *See* Increment, 66, 721
- +=. See*
  - Add and assign, 1089
  - Move forward, 1101
  - string** (add at end), 851, 1176
- ,* *(comma). See*
  - Comma operator, 1090
  - List separator, 1103, 1122–1123
- .* *See*
  - Minus (subtraction), 66, 1088
  - Regular expression (range), 877
- . See* Decrement, 66, 1087, 1141
- > (arrow). See* Member access, 608, 1087, 1109, 1141
- = See*
  - Move backward, 1101, 1142
  - Subtract and assign, 67, 1090
- . (dot). See*
  - Member access, 306, 607–608, 1086–1087
  - Regular expression, 872, 1178
- ... (ellipsis). See*
  - Arguments (unchecked), 1105–1106
  - Catch all exceptions, 152
- /.* *See* Divide, 66, 1088
- //. See* Line comment, 45
- /\*...\*/. See* Block comment, 238
- /=. See* Divide and assign, 67, 1090
- : (colon). See*
  - Base and member initializers, 315, 477, 555
  - Conditional expression, 268
  - Label, 106–108, 306, 511, 1096
- ::. See* Scope (resolution), 295, 314, 1083, 1086
- ;* *(semicolon). See* Statement (terminator), 50, 100

- <. See** Less than, 67, 1088
  - <<. See** Bitwise logical operations (left shift), 956, 1088  
Output, 363–365, 1173
  - <=. See** Less than or equal, 67, 1088
  - <<=. See** Bitwise logical operations (shift left and assign), 1090
  - < . . >. See** Template (arguments and parameters), 153, 678–679
  - =. See** Assignment, 66, 1090  
Initialization, 69–73, 1219
  - ==. See** Equal, 67, 1088
  - >. See** Greater than, 67, 1088  
Input prompt, 223  
Template (argument-list terminator), 679
  - >=. See** Greater than or equal, 67, 1088
  - >>. See** Bitwise logical operations (right shift), 956, 1088  
Input, 61, 365
  - >>=. See** Bitwise logical operations (shift right and assign), 1090
  - ?.** See Conditional expression, 268, 1089  
Regular expression, 867–868, 873, 874–875, 1178
  - [], See** Array of (in declaration), 649, 1099  
Regular expression (character class), 872, 1178  
Subscripting, 594, 649, 1101
  - \ (backslash). See** Character literal, 1079–1080  
Escape character, 1178  
Regular expression (escape character), 866–867, 873, 877
  - ^.** See Bitwise logical operations (exclusive or), 956, 1089, 1094  
Regular expression (not), 873, 1178
  - ^=.** See Bitwise logical operations (xor and assign), 1090
  - \_.** See Underscore, 75, 76, 1081
  - {}.** See Block delimiter, 47, 111  
Initialization, 83  
List, 83  
Regular expression (range), 867, 873–875, 1178
  - |.** See Bitwise logical operations (bitwise or), 956, 1089, 1094  
Regular expression (or), 867–868, 873, 876, 1178
  - |=. See** Bitwise logical operations (or and assign), 1090
  - ||. See** Logical or, 1089, 1094
  - ~.** See Bitwise logical operations (complement), 956, 1087  
Destructors, 601–603
  - 0 (zero). See** Null pointer, 598  
Prefix, 382, 384  
**printf()** format specifier, 1188–1189
  - 0x.** See Prefix, 382, 384
- ## A
- a**, append file mode, 1186
  - \a** alert, character literal, 1079
  - abort()**, 1194–1195
  - abs()**, absolute value, 917, 1181
  - complex**, 920, 1183
  - Abstract classes, 495, 1217  
class hierarchies, 512  
creating, 495, 512, 1118–1119  
**Shape** example, 495–496
  - Abstract-first approach to programming, 10
  - Abstraction, 92–93, 1217  
level, ideals, 812–813
  - Access control, 306, 505, 511  
base classes, 511  
encapsulation, 505  
members, 492–493  
private, 505, 511  
private by default, 306–307  
**private:** label, 306  
private *vs.* public, 306–308  
protected, 505, 511  
**protected:** label, 511  
public, 306, 505, 511  
public by default, 307–308. *See also struct*
  - public:** label, 306
  - Shape** example, 496–499
  - accumulate()**, 759, 770–772, 1183  
accumulator, 770  
generalizing, 772–774
  - acos()**, arccosine, 917, 1182

Action, 47  
Activation record, 287. *See also* Stacks  
Ada language, 832–833  
Adaptors  
    **bind()**, 1164  
    container, 1144  
    function objects, 1164  
    **mem\_fn()**, 1164  
    **not1()**, 1164  
    **not2()**, 1164  
    **priority\_queue**, 1144  
    **queue**, 1144  
    **stack**, 1144  
    **add()**, 449–450, 491–492, 615–617  
Add (plus) **+**, 66, 1088  
Add and assign **+=**, 66, 73, 1090  
Additive operators, 1088  
Address, 588, 1217  
    unchecked conversions, 943–944  
Address of (unary) **&**, 588, 1087  
Ad hoc polymorphism, 682–683  
    **adjacent\_difference()**, 770, 1184  
    **adjacent\_find()**, 1153  
    **advance()**, 615–617, 739, 1142  
Affordability, software, 34  
Age distribution example, 538–539  
Alert markers, 3  
Algol60 language, 827–829  
Algol family of languages, 826–829  
    **<algorithm>**, 759, 1133  
Algorithms, 1217  
    and containers, 722  
    header files, 1133–1134  
    numerical, 1183–1184  
    passing arguments to. *See* Function objects  
Algorithms, numerical, 770, 1183–1184  
    **accumulate()**, 759, 770–774, 1183  
    **adjacent\_difference()**, 770, 1184  
    **inner\_product()**, 759, 770, 774–776, 1184  
    **partial\_sum()**, 770, 1184  
Algorithms, STL, 1152–1153  
    **<algorithm>**, 759  
    **binary\_search()**, 796  
    comparing elements, 759  
    **copy()**, 758, 789–790  
    **copy\_if()**, 789  
    copying elements, 758  
    **count()**, 758  
    **count\_if()**, 758  
    **equal()**, 759  
    **equal\_range()**, 758, 796  
    **find()**, 758, 759–763  
    **find\_if()**, 758, 763–764  
    heap, 1160  
    **lower\_bound()**, 796  
    **max()**, 1161  
    **merge()**, 758  
    merging sorted sequences, 758  
    **min()**, 1161  
    modifying sequence, 1154–1156  
    mutating sequence, 1154–1156  
    nonmodifying sequence, 1153–1154  
    numerical. *See* Algorithms, numerical  
    permutations, 1160–1161  
    **search()**, 795–796  
    searching, 1157–1159. *See also* **find\_if()**; **find()**  
    set, 1159–1160  
    **shuffle()**, 1155–1156  
    **sort()**, 758, 794–796  
    sorting, 758, 794–796, 1157–1159  
    summing elements, 759  
    testing, 1001–1008  
    **unique\_copy()**, 758, 789, 792–793  
    **upper\_bound()**, 796  
    utility, 1157  
    value comparisons, 1161–1162  
Aliases, 1128, 1217. *See also* References  
Allocating memory. *See also* Deallocating memory;  
    Memory  
    **allocator\_type**, 1147  
    **bad\_alloc** exception, 1094  
    C++ and C, 1043–1044  
    **calloc()**, 1193  
    embedded systems, 935–936, 940–942  
    free store, 593–594  
    **malloc()**, 1043–1044, 1193  
    **new**, 1094–1095  
    pools, 940–941  
    **realloc()**, 1045  
    stacks, 942–943  
    **allocator\_type**, 1147  
Almost containers, 751, 1145  
    **alnum**, **regex** character class, 878, 1179  
    **alpha**, **regex** character class, 878, 1179  
Alternation  
    patterns, 194  
    regular expressions, 876  
Ambiguous function call, 1104  
Analysis, 35, 176, 179  
**and**, synonym for **&**, 1037, 1038

**and\_eq**, synonym for `&=`, 1037, 1038  
**app** mode, 389, 1170  
**append()**, 851, 1177  
Append  
  files, 389, 1186  
  string `+=`, 851  
Application  
  collection of programs, 1218  
  operator `O`, 766  
Approximation, 532–537, 1218  
Arccosine, **acos()**, 917  
Arcsine, **asin()**, 918  
Arctangent, **atan()**, 918  
**arg()**, of complex number, theta, 920, 1183  
Argument deduction, 689–690  
Argument errors  
  callee responsibility, 143–145  
  caller responsibility, 142–143  
  reasons for, 144–145  
Arguments, 272, 1218  
  formal. *See* Parameters  
  functions, 1105–1106  
  passing. *See* Passing arguments  
  program input, 91  
  source of exceptions, 147–148  
  templates, 1122–1123  
  types, class interfaces, 324–326  
  unchecked, 1029–1030, 1105–1106  
  unexpected, 136  
Arithmetic if `?:`, 268. *See also* Conditional expression `?:`  
Arithmetic operations. *See* Numerics  
**<array>**, 1133  
Arrays, 648–650, 1218. *See also* Containers; **vector**  
  declaration, 649  
  dereferencing, 649  
  accessing elements, 649, 899–901  
  assignment, 653–654  
  associative. *See* Associative containers  
  built-in, 747–749  
  copying, 653–654  
  C-style strings, 654–655  
  dereferencing, 649  
  element numbering, 649  
  initializing, 596–598, 654–656  
  multidimensional, 895–897, 1102  
  palindrome example, 660–661  
  passing pointers to arrays, 944–951  
  pointers to elements, 650–652  
  range checking, 649  
  subscripting `[I]`, 649  
  terminating zero, 654–655  
  **vector** alternative, 947–951  
Arrays and pointers, 651–658  
  debugging, 656–659  
**array** standard library class, 747–749, 1144  
**asin()**, arcsine, 918, 1182  
**asm()**, assembler insert, 1037  
Assemblers, 820  
Assertions  
  **assert()**, 1061  
  **<cassert>**, 1135  
  debugging, 163  
  definition, 1218  
**assign()**, 1148  
Assignment `=`, 69–73  
  arrays, 653–654  
  assignment and initialization, 69–73  
  composite assignment operators, 73–74  
  containers, 1148  
**Date** example, 309–310  
enumerators, 318–319  
expressions, 1089–1090  
**string**, 851  
**vector**, resizing, 675–677  
Assignment operators (composite), 66  
  `%=`, 73, 1090  
  `&=`, 1090  
  `*=`, 73, 1089  
  `+=`, 73, 1090, 1141  
  `-=`, 73, 1090, 1142  
  `/=`, 73, 1090  
  `<<=`, 1090  
  `>>=`, 1090  
  `^=`, 1090  
  `|=`, 1090  
Associative arrays. *See* Associative containers  
Associative containers, 776, 1144  
  email example, 856–860  
  header files, 776  
  map, 776  
  **multimap**, 776, 860–861  
  **multiset**, 776  
  operations, 1151–1152  
  **set**, 776  
  **unordered\_map**, 776  
  **unordered\_multimap**, 776  
  **unordered\_multiset**, 776  
  **unordered\_set**, 776

Assumptions, testing, 1009–1011  
**at0**, range-checked subscripting, 693–694,  
    1149  
**atan0**, arctangent, 918, 1182  
**ate** mode, 389, 1170  
**atof0**, string to **double**, 1192  
**atoi0**, string to **int**, 1192  
**atol0**, string to **long**, 1192  
AT&T Bell Labs, 838  
AT&T Labs, 838  
**attach0** vs. **add0** example, 491–492  
**auto**, 732–734, 760  
Automatic storage, 591–592, 1083. *See also*  
    Stack storage  
**Axis** example, 424–426, 443, 529–532,  
    543–546

## B

**b**, binary file mode, 1186  
Babbage, Charles, 832  
**back()**, last element, 737, 1149  
**back\_inserter()**, 1162  
Backus, John, 823  
Backus-Naur (BNF) Form, 823, 828  
**bad\_alloc** exception, 1094  
**bad0** stream state, 355, 1171  
Base-2 number system (binary), 1078–1079  
Base-8 number system (octal), 1077–1078  
Base-10  
    logarithms, 918  
    number system (decimal), 1077–1078  
Base-16 number system (hexadecimal),  
    1077–1078  
Balanced trees, 780–782  
Base and member initializers, 315, 477, 555  
Base classes, 493–496, 504–507, 1218  
    abstract classes, 495, 512–513, 1118–1119  
    access control, 511  
    derived classes, 1116–1117  
    description, 504–506  
    initialization of, 477, 555, 1113, 1117  
    interface, 513–514  
    object layout, 506–507  
    overriding, 508–511  
**Shape** example, 495–496  
virtual function calls, 501, 506–507  
**vptr**, 506  
**vtbl**, 506  
Base-e exponentials, 918

**basic\_string**, 852  
Basic guarantee, 702  
BCPL language, 838  
**begin()**  
    iterator, 1148  
    **string**, 851, 1177  
    **vector**, 721  
Bell Telephone Laboratories (Bell Labs), 836,  
    838–842, 1022–1023  
Bentley, John, 933, 966  
Bidirectional iterator, 1142  
**bidirectional** iterators, 752  
Big-O notation, complexity, 785  
Binary I/O, 390–393  
**binary** mode, 389, 1170  
Binary number system, 1078–1079  
Binary search, 758, 779, 795–796  
**binary\_search0**, 796, 1158  
**bind()** adaptor, 1164  
**bitand**, synonym for **&**, 1037, 1038  
Bitfields, 956–957, 967–969, 1120–1121  
**bitor**, synonym for **|**, 1038  
Bits, 78, 954, 1218  
    bitfields, 956–957  
    **bool**, 955  
    **char**, 955  
    enumerations, 956  
    integer types, 955  
    manipulating, 965–967  
    signed, 961–965  
    size, 955–956  
    **unsigned**, 961–965  
**<bitset>**, 1133  
**bitset**, 959–961  
    bitwise logical operations, 960  
    construction, 959  
    exceptions, 1138  
    I/O, 960  
Bitwise logical operations, 956–959, 1094  
and **&**, 956–957, 1089, 1094  
or **|**, 956, 1089, 1094  
or and assign, **|=**, 966  
and and assign **&=**, 1090  
complement **~**, 956  
exclusive or **^**, 956, 1089, 1094  
exclusive or and assign **^=**, 1089  
left shift **<<**, 956  
left shift and assign **<<=**, 1089  
right shift **>>**, 956  
right shift and assign **>>=**, 1089

- Blackboard, 36  
 Black-box testing, 992–993  
**blank**, character class, `regex`, 878, 1179  
 Block, 111  
   debugging, 161  
   delimiter, 47, 111  
   nesting within functions, 271  
   **try** block, 146–147  
 Block comment `/*...*/`, 238  
 Blue marginal alerts, 3  
 BNF (Backus-Naur) Form, 823, 828  
 Body, functions, 114  
**bool**, 63, 66–67, 1099  
   bits in memory, 78  
   bit space, 955  
   C++ and C, 1026, 1038  
   size, 78  
**boolalpha**, manipulator, 1173  
 Boolean conversions, 1092  
 Borland, 831  
 Bottom-up approach, 9, 811  
 Bounds error, 149  
 Branching, testing, 1006–1008. *See also*  
   Conditional statements  
**break**, **case** label termination, 106–108  
 Broadcast functions, 903  
**bsearch()**, 1194–1195  
 Buffer, 348  
   flushing, 240–241  
   *iostream*, 406  
   overflow, 661, 792, 1006. *See also* `gets()`, `scanf()`  
 Bugs, 158, 1218. *See also* Debugging; Testing  
   finding the last, 166–167  
   first documented, 824–825  
   regression tests, 993  
 Built-in types, 304, 1099  
   arrays, 747–749, 1101–1102  
   **bool**, 77, 1100  
   characters, 77, 891, 1100  
   default constructors, 328  
   exceptions, 1126  
   floating-point, 77, 891–895, 1100  
   integers, 77, 891–895, 961–965, 1100  
   pointers, 588–590, 1100–1101  
   references, 279–280, 1102–1103  
**Button** example, 443, 561–563  
   attaching to menus, 571  
   detecting a click, 557  
 Byte, 78, 1218  
   operations, C-style strings, 1048–1049
- C**
- .**c** suffix, 1029  
.b.**pp**, suffix, 48, 1200  
 C# language, 831  
 C++ language, 839–842. *See also* Programming;  
   Programs; Software  
   coding standards, list of, 983  
   portability, 11  
   use for teaching, xxiv, 6–9  
 C++ and C, 1022–1024  
   C functions, 1028–1032  
   C linkage convention, 1033  
   C missing features, 1025–1027  
   calling one from the other, 1032–1034  
   casts, 1040–1041  
   compatibility, 1024–1025  
   **const**, 1054–1055  
   constants, 1054–1055  
   container example, 1059–1065  
   definitions, 1038–1040  
   **enum**, 1042  
   family tree, 1023  
   free-store, 1043–1045  
   input/output, 1050–1054  
   keywords, 1037–1038  
   layout rules, 1034  
   macros, 1054–1059  
   **malloc()**, 1043–1044  
   namespaces, 1042–1043  
   nesting **structs**, 1037  
   old-style casts, 1040  
   opaque types, 1060  
   performance, 1024  
   **realloc()**, 1045  
   structure tags, 1036–1037  
   type checking, 1032–1033  
   **void**, 1030  
   **void\***, 1041–1042  
   “C first” approach to programming, 9  
 C language, 836–839. *See also* C standard  
   library  
   C++ compatibility, 1022–1024. *See also*  
     C++ and C  
   K&R, 838, 1022–1023  
   linkage convention, 1033  
   missing features, 1025–1027  
 C standard library  
   C-style strings, 1191  
   header files, 1135

input/output. *See* C-style I/O (stdio)  
memory, 1192–1193  
C-style casts, 1040–1041, 1087, 1095  
C-style I/O (stdio)  
    %, conversion specification, 1187  
    conversion specifications, 1188–1189  
    file modes, 1186  
    files, opening and closing, 1186  
    *fprintf()*, 1051–1052, 1187  
    *getc()*, 1052, 1191  
    *getchar()*, 1045, 1052–1053, 1191  
    *gets()*, 1052, 1190–1191  
    output formats, user-defined types, 1189–1190  
    padding, 1188  
    *printf()*, 1050–1051, 1187  
    *scanf()*, 1052–1053, 1190  
    *stderr*, 1189  
    *stdin*, 1189  
    *stdout*, 1189  
    truncation, 1189  
C-style strings, 654–655, 1045–1047, 1191  
    byte operations, 1048–1049  
    *const*, 1047–1048  
    copying, 1046–1047, 1049  
    executing as a command, *system()*, 1194  
    lexicographical comparison, 1046  
    operations, 1191–1192  
    pointer declaration, 1049–1050  
    *strcat()*, concatenate, 1047  
    *strchr()*, find character, 1048  
    *strcmp()*, compare, 1046  
    *strcpy()*, copy, 1047, 1049  
    from *string*, *c\_str()*, 350, 851  
    *strlen()*, length of, 1046  
    *strncat()*, 1047  
    *strncmp()*, 1047  
    *strncpy()*, 1047  
    three-way comparison, 1046  
CAD/CAM, 27, 34  
Calculator example, 174, 186–188  
    analysis and design, 176–179  
    *expression()*, 197–200  
    *get\_token()*, 196  
    grammars and programming, 188–195  
    parsing, 190–193  
    *primary()*, 196, 208  
    symbol table, 247  
    *term()*, 196, 197–202, 206–207  
Token, 185–186  
*Token\_stream*, 206–214, 240–241  
Call stack, 290  
Callback functions, 556–559  
Callback implementation, 1208–1209  
Calling functions. *See* Function calls  
*calloc()*, 1193  
Cambridge University, 839  
*capacity()*, 673–674, 1151  
Capital letters. *See* Case (of characters)  
Case (of characters)  
    formatting, 397–398  
    identifying, 397  
    *islower()*, 397, 1175  
    *map* container, 782  
    in names, 74–77  
    sensitivity, 397–398  
    *tolower()*, changing case, 398, 1176  
    *toupper()*, changing case, 398, 1176  
case labels, 106–108  
*<cassert>*, 1135  
Casting away *const*, 609–610  
Casts. *See also* Type conversion  
    C++ and C, 1026, 1038  
    casting away *const*, 609  
    *const\_cast*, 1095  
    C-style casts, 1040–1041  
    *dynamic\_cast*, 932, 1095  
    *lexical\_cast* example, 855  
    *narrow\_cast* example, 153  
    *reinterpret\_cast*, 609  
    *static\_cast*, 609, 944, 1095  
    unrelated types, 609  
CAT scans, 30  
*catch*, 147, 1038  
Catch all exceptions .., 152  
Catching exceptions, 146–153, 239–241, 1126  
*cb\_next()* example, 556–559  
*<cctype>*, 1135, 1175  
*ceil()*, 917, 1181  
*cerr*, 151, 1169, 1189  
*<cerrno>*, 1135  
*<cfloat>*, 1135  
Chaining operations, 180–181  
Character classes  
    list of, 1179  
    in regular expressions, 873–874, 878  
Character classification, 397–398, 1175–1176  
Character literals, 161, 1079–1080  
*CHAR\_BIT* limit macro, 1181  
*CHAR\_MAX* limit macro, 1181  
*CHAR\_MIN* limit macro, 1181

**char** type, 63, 66–67, 78  
 bits, 955  
 built-in, 1099  
 properties, 741–742  
**signed** *vs.* **unsigned**, 894, 964  
**cin**, 61  
 C equivalent. *See* **stdin**  
 standard character input, 61, 347, 1169  
**Circle** example, 469–472, 497  
*vs.* **Ellipse**, 474  
 Circular reference. *See* Reference (circular)  
**class**, 183, 1036–1037  
 Class  
 abstract, 495, 512–513, 1118–1119. *See also*  
   Abstract classes  
 base, 504–506  
 coding standards, 981  
 concrete, 495–496, 1218  
**const** member functions, 1110  
 constructors, 1112–1114, 1119–1120  
 copying, 1115, 1119  
 creating objects. *See* Concrete classes  
 default constructors, 327–330  
 defining, 212, 305, 1108, 1218  
 derived, 504  
 destructors, 1114–1115, 1119  
 encapsulation, 505  
**friend** declaration, 1111  
 generated operations, 1119–1120  
 grouping related, 512  
 hierarchies, 512  
 history of, 834  
 implementation, 306–308  
 inheritance, 504–505, 513–514  
 interface, 513–514  
 member access. *See* Access control  
 naming. *See* Namespaces  
 nesting, 270  
 object layout, 506–507  
 organizing. *See* Namespaces  
 parameterized, 682–683. *See also* Template  
**private**, 306–308, 505, 511, 1108–1109  
**protected**, 495, 505, 511  
**public**, 306–308, 505, 511, 1108–1109  
 run-time polymorphism, 504–505  
 subclasses, 504. *See also* Derived classes  
 superclasses, 504. *See also* Base classes  
 templates, 681–683  
**this** pointer, 1110  
 types as parameters. *See* Template  
**union**, 1121  
 unqualified name, 1110  
 uses for, 305  
 Class interfaces, 323, 1108  
 argument types, 324–326  
**const** member functions, 330–332  
 constants, 330–332. *See also* **const**  
 copying, 326–327  
 helper functions, 332–334  
 immutable values, 330–332  
 initializing objects, 327–330  
 members, 332–334  
 mutable values, 332–334  
 public *vs.* private, 306–308  
 symbolic constants, defining, 326  
 uninitialized variables, 327–330  
 Class members, 305, 1108  
 . (dot), 306, 1109  
 :: (scope resolution), 1109  
 accessing, 306. *See also* Access control  
 allocated at same address, 1121  
 bitfields, 1120–1121  
 in-class definition, 1112  
 class interfaces, 332–334  
 data, 305  
 definitions, 1112  
 function, 314–316  
 out-of-class definition, 1112  
**Token\_stream** example, 212  
**Token** example, 183–184  
 Class scope, 267, 1083  
 Class template  
 parameterized class, 682–683  
 parameterized type, 682–683  
 specialization, 681  
 type generators, 681  
**classic\_elimination()** example, 910–911  
 Cleaning up code  
 comments, 237–238  
 functions, 234–235  
 layout, 235–236  
 logical separations, 234–235  
 revision history, 237–238  
 scaffolding, 234–235  
 symbolic constants, 232–234  
**clear()**, 355–358, 1150  
**<limits>**, 1135  
**<locale>**, 1135

- clock()**, 1015–1016, 1193  
**clock\_t**, 1193  
**clone()** example, 504  
**Closed\_polyline** example, 456–458  
    *vs.* **Polygon**, 458  
**close()** file, 352  
**<cmath>**, 918, 1135, 1182  
**cntrl**, 878, 1179  
COBOL language, 823–825  
Code  
    definition, 1218  
    layout, cleaning up, 235–236  
    libraries, uses for, 177  
    storage, 591–592  
    structure, ideals, 810–811  
    test coverage, 1008  
Coding standards, 974–975  
    C++, list of, 983  
    complexity, sources of, 975  
    ideals, 976–977  
    sample rules, 977–983  
**Color** example, 425–426, 450–452  
    color chat example, 465–467  
    fill, 431–432, 462–464, 500  
    transparency, 451  
Columns, matrices, 900–901, 906  
Command-line, 47  
Comments, 45–46  
    block **/\*...\*/**, 238, 1076  
    C++ and C, 1026  
    cleaning up, 237–238  
    *vs.* code, 238  
    line **//**, 45–46, 1076  
    role in debugging, 159–160  
Common Lisp language, 825  
Communication skills, programmers, 22  
Compacting garbage collection, 938–939  
Comparison, 67. *See also* **<; ==**  
    C-style strings, 1045–1047  
    characters, 740  
    containers, 1151  
    **key\_compare**, 1147  
    lexicographical, C-style strings, 1046  
    **lexicographical\_compare()**, 1162  
    **min/max** algorithms, 1161–1162  
    **string**, 851  
    three-way, 1046  
Compatibility. *See* C++ and C  
Compile-time errors. *See* Errors, compile-time  
Compiled languages, 47–48  
Compilers, 48, 1218  
    compile-time errors, 51  
    conditional compilation, 1058–1059  
    syntax checking, 48–50  
**compl**, synonym for **~**, 1037, 1082  
**complex**  
     $*$ , multiply, 919, 1183  
     $+$ , add (plus), 919, 1183  
     $\ll$ , output, 1183  
     $\neq$ , not equal (inequality), 919, 1183  
     $=$ , equal, 919, 1183  
     $\gg$ , input, 920, 1183  
     $/$ , divide, 919, 1183  
     $\ll$ , output, 920  
    **abs()**, absolute value, 920, 1183  
    **conj()**, conjugate, 920  
    Fortran language, 920  
    **imag()**, imaginary part, 920  
    **norm()**, square of **abs()**, 919  
    number types, 1182–1183  
    **polar()**, polar coordinate, 920  
    **real()**, real part, 920  
    rho, 920  
    square of **abs()**, 919  
    theta, 920  
**<complex>**, 1134  
**complex** operators, 919–920, 1183  
    standard math functions, 1181  
Complex numbers, 919–920  
Complexity, 1218  
    sources of, 975  
Composite assignment operators, 73–74  
Compound statements, 111  
Computation, 91. *See also* Programs; Software  
    correctness, 92–94  
    data structures, 90–91  
    efficiency, 92–94  
    input/output, 91  
    objectives, 92–94  
    organizing programs, 92–94  
    programmer ideals, 92–94  
    simplicity, 92–94  
    state, definition, 90–91  
Computation *vs.* data, 717–720  
Computer-assisted surgery, 30  
Computers  
    CAT scans, 30  
    computer-assisted surgery, 30

Computers, *continued*  
 in daily life, 19–21  
 information processing, 32  
 Mars Rover, 33  
 medicine, 30  
 pervasiveness of, 19–21  
 server farms, 31–32  
 shipping, 26–28  
 space exploration, 33  
 telecommunications, 28–29  
 timekeeping, 26  
 world total, 19

Computer science, 12, 24–25

Concatenation of **strings**, 66  
**+**, 68–69, 851, 1176  
**+=**, 68–69, 851, 1176

Concept-based approach to programming, 6

Concrete classes, 495–496, 1218

Concrete-first approach to programming, 6

Concurrency, 932

Conditional compilation, 1058–1059

Conditional expression **?:**, 268, 1089

Conditional statements. *See also* Branching,  
 testing  
**for**, 111–113  
**if**, 102–104  
**switch**, 105–109  
**while**, 109–111

Conforming programs, 1075

Confusing variable names, 77

**conj()**, complex conjugate, 920, 1183

Conjugate, 920

Consistency, ideals, 814–815

Console, as user interface, 552

Console input/output, 552

Console window, displaying, 162

**const**, 95–97. *See also* Constant; Static storage;  
**static const**  
 C++ and C, 1026, 1054–1055  
 class interfaces, 330–332  
 C-style strings, 1047–1048  
 declarations, 262–263  
 initializing, 262  
 member functions, 330–332, 1110  
 overloading on, 647–648  
 passing arguments by, 276–278, 281–284  
 type, 1099  
**\*const**, immutable pointer, 1099

Constant. *See also* **const**, expressions, 1093

**const\_cast**, casting away **const**, 609, 1095

**const\_iterator**, 1147

**constexpr**, 96–97, 290–291, 1093, 1104

Constraints, **vector** range checking, 695

Constructors, 310–312, 1112–1114. *See also*  
 Constructors; Initialization  
 containers, 1148  
 copy, 633–634, 640–646  
**Date** example, 311  
**Date** example 307, 324–326  
 debugging, 643–646  
 default, 327–330, 1119  
 error handling, 313, 700–702  
 essential operations, 640–646  
 exceptions, 700–702  
**explicit**, 642–643  
 implicit conversions, 642–643  
 initialization of bases and members, 315, 477,  
 555  
 invariant, 313–314, 701–702  
 move, 637–640  
 need for default, 641  
**Token** example, 184

Container adaptors, 1144

Containers, 148, 749–751, 1218. *See also* Arrays;  
**list**; **map**, associative array; **vector**  
 and algorithms, 722  
 almost containers, 751, 1145  
 assignments, 1148  
 associative, 1144, 1151–1152  
**capacity()**, 1150–1151  
 of characters. *See* **string**  
 comparing, 1151  
 constructors, 1148  
 contiguous storage, 741  
 copying, 1151  
 destructors, 1148  
 element access, 1149  
 embedded systems, 951–954  
 header files, 1133–1134  
 information sources about, 750  
 iterator categories, 752  
 iterators, 1148  
 list operations, 1150  
 member types, 1147  
 operations overview, 1146–1147  
 queue operations, 1149  
 sequence, 1144  
**size()**, 1150  
 stack operations, 1149  
 standard library, 1144–1152

swapping, 1151  
templates, 686–687  
Contents of `*` (dereference, indirection), 594  
Contiguous storage, 741  
Control characters, `iscntrl()`, 397  
Control inversion, GUIs, 569–570  
Control variables, 110  
Controls. *See* `Widget` example  
Conversion specifications, `printf()`, 1188–1189  
Conversion. *See also* Type conversion  
    character case, 398  
    representation, 374–376  
    unchecked, 943–944  
Coordinates. *See also* `Point` example  
    computer screens, 419–420  
    graphs, 426–427  
**`copy()`**, 789–790, 1154  
Copy assignments, 634–636, 640–646  
Copy constructors, 633–634, 640–646  
**`copy_backward()`**, 1154  
**`copy_if()`**, 789  
Copying, 631–637  
    arrays, 653–654  
    class interfaces, 326–327  
    containers, 1151  
    C-style strings, 1046–1047, 1049  
    I/O streams, 790–793  
    objects, 503–504  
    sequences, 758, 789–794  
**`vector`**, 631–636, 1148  
Correctness  
    definition, 1218  
    ideals, 92–94, 810  
    importance of, 929–930  
    software, 34  
**`cos()`**, cosine, 527–528, 917, 1181  
**`cosh()`**, hyperbolic cosine, 1182  
Cost, definition, 1219  
**`count()`**, 758, 1154  
**`count_if()`**, 758, 1154  
**`cout`**, 45  
    C equivalent. *See* `stdout`  
    printing error messages, 151. *See also* `cerr`  
    standard output, 347, 1169  
Critical systems, coding standards, 982–983  
**`<cstddef>`**, 1136  
**`<cstdio>`**, 1135  
**`<cstdlib>`**, 1135, 1193, 1194  
**`c_str()`**, 1177  
**`<cstring>`**, 1135, 1175, 1193  
**`<ctime>`**, 1135, 1193  
Ctrl D, 124  
Ctrl Z, 124  
Current object, 317. *See also* `this` pointer  
Cursor, definition, 45  
**`<cwchar>`**, 1136  
**`<cwctype>`**, 1136

## D

**d**, any decimal digit, `regex`, 878, 1179  
**d**, decimal digit, `regex`, 873, 1179  
**D**, not a decimal digit, `regex`, 873, 1179  
**d** suffix, 1079  
Dahl, Ole-Johan, 833–835  
Data. *See also* Containers; Sequences; `list`; `map`,  
    associative array; `vector`  
abstraction, 816  
collections. *See* Containers  
    *vs.* computation, 717–720  
    generalizing code, 714–716  
    in memory. *See* Free store (heap storage)  
    processing, overview, 712–716  
    separating from algorithms, 722  
    storing. *See* Containers  
    structure. *See* Containers; `class`; `struct`  
    traversing. *See* Iteration; Iterators  
    uniform access and manipulation, 714–716. *See*  
        *also* STL (Standard Template Library)  
Data member, 305, 492–493  
Data structure. *See* Data; `struct`  
Data type. *See* Type  
Date and time, 1193–1194  
**Date** example, *See* Chapters 6–7  
Deallocating memory, 598–600, 1094–1095. *See*  
    *also* `delete[]`; `delete`  
Debugging, 52, 158, 1219. *See also* Errors; Testing  
    arrays and pointers, 656–659  
    assertions, 163  
    block termination, 161  
    bugs, 158  
    character literal termination, 161  
    commenting code, 159–160  
    compile-time errors, 161  
    consistent code layout, 160  
    constructors, 643–646  
    declaring names, 161  
    displaying the console window, 162  
    expression termination, 161  
    finding the last bug, 166–167

Debugging, *continued*  
 function size, 160  
 GUIs, 575–577  
 input data, 166  
 invariants, 162–163  
 keeping it simple, 160  
 logic errors, 154–156  
 matching parentheses, 161  
 naming conventions, 160  
 post-conditions, 165–166  
 pre-conditions, 163–165  
 process description, 158–159  
 reporting errors, 159  
 stepping through code, 162  
 string literal termination, 161  
 systematic approach, 166–167  
 test cases, 166, 227  
 testing, 1012  
 tracing code execution, 162–163  
 transient bugs, 595  
 using library facilities, 160  
 widgets, 576–577

**dec** manipulator, 382–383, 1174

Decimal digits, **isDigit()**, 397

Decimal integer literals, 1077

Decimal number system, 381–383, 1077–1078

Deciphering (decryption), example, 969–974

Declaration operators, 1099  
 & reference to, 276–279, 1099  
 \* pointer to, 587, 1099  
 [] array of, 649, 1099  
 () function of, 113–115, 1099

Declarations, 51, 1098–1099  
 C++ and C, 1026  
 classes, 306  
 collections of. *See* Header files  
 constants, 262–263  
 definition, 51, 77, 257, 1098–1099, 1219  
*vs.* definitions, 259–260  
 entities used for, 261  
**extern** keyword, 259  
 forward, 261  
 function, 257–258, 1103  
 function arguments, 272–273  
 function return type, 272–273  
 grouping. *See* Namespaces  
 managing. *See* Header files  
 need for, 261  
 order of, 215

parts of, 1098  
 subdividing programs, 260–261  
 uses for, 1098  
 variables, 260, 262–263

Decrementing --, 97  
 iterator, 1141–1142  
 pointer, 652

Deep copy, 636

Default constructors, 328–329  
 alternatives for, 329–330  
 for built-in types, 328  
 initializing objects, 327  
 need for, identifying, 641  
 uses for, 328–329

**#define**, 1129

Definitions, 77, 258–259, 1219. *See also*  
 Declarations  
 C++ and C, 1038–1040  
*vs.* declarations, 259–260  
 function, 113–115, 272–273

**delete**  
 C++ and C, 1026, 1037  
 deallocating free store, 1094–1095  
 destructors, 601–605  
 embedded systems, 932, 936–940  
 free-store deallocation, 598–600  
 in unary expressions, 1087  
**delete[]**, 599, 1087, 1094–1095

Delphi language, 831

Dependencies, testing, 1002–1003

Depth-first approach to programming, 6

**deque**, double ended queue, 1144  
**<deque>**, 1133

Dereference/indirection  
 \*, 594. *See also* Contents of  
 [], 118. *See also* Subscripting

Derivation, classes, 505

Derived classes, 505, 1219  
 access control, 511  
 base classes, 1116–1117  
 inheritance, 1116–1117  
 multiple inheritance, 1117  
 object layout, 506–507  
 overview, 504–506, 1116–1117  
**private** bases and members, 511  
**protected** bases and members, 511  
**public** bases and members, 511  
 specifying, 507–508  
 virtual functions, 1117–1118

- Design, 35, 176, 179, 1219  
Design for testing, 1011–1012  
Destructors, 601–603, 1114–1115, 1219. *See also Constructors*  
    containers, 1148  
    debugging, 643–646  
    default, 1119  
    essential operations, 640–646  
    exceptions, 700–702  
    freeing resources, 323, 700–702  
    and free store, 604–605  
    generated, 603  
    RAII, 700–702  
    virtual, 604–605  
    where needed, 641–642  
Device drivers, 346  
Dictionary examples, 123–125, 788  
**difference\_type**, 1147  
**digit**, character class, 878, 1179  
Digit, word origin, 1077  
Dijkstra, Edsger, 827–828, 992  
Dimensions, matrices, 898–901  
Direct expression of ideas, ideals, 811–812  
Dispatch, 504–505  
Display model, 413–414  
**distance()**, 1142  
Divide **/**, 66, 1088  
Divide and assign **/=**, 67, 1090  
Divide and conquer, 93  
Divide-by-zero error, 201–202  
**divides()**, 1164  
Domain knowledge, 934  
Dot product. *See inner\_product()*  
**double** floating-point type, 63, 66–67, 78, 1099  
Doubly-linked lists, 613, 725. *See also list*  
**draw()** example  
    fill color, 500  
    line visibility, 500  
    **Shape**, 500–502  
**draw\_lines()** example. *See also draw() example*  
    **Closed\_polyline**, 458  
    **Marked\_polyline**, 475–476  
    **Open\_polyline**, 456  
    **Polygon**, 459  
    **Rectangle**, 465  
    **Shape**, 500–502  
duration..., 1016, 1185  
**duration\_cast**, 1016, 1185  
Dynamic dispatch, 504–505. *See also Virtual functions*  
Dynamic memory, 935–936, 1094. *See also Free store (heap storage)*  
**dynamic\_cast**, type conversion, 1095  
    exceptions, 1138  
predictability, 932

## E

- Efficiency  
    ideals, 92–94, 810  
    **vector** range checking, 695  
Einstein, Albert, 815  
Elements. *See also vector*  
    numbering, 649  
    pointers to, 650–652  
    variable number of, 649  
**Ellipse** example, 472–474  
    *vs. Circle*, 474  
Ellipsis ...  
    arguments (unchecked), 1105–1106  
    catch all exceptions, 152  
**else**, in **if**-statements, 102–104  
Email example, 855–865  
Embedded systems  
    coding standards, 975–977, 983  
    concurrency, 932  
    containers, 951–954  
    correctness, 929–930  
    **delete** operator, 932  
    domain knowledge, 934  
    **dynamic\_cast**, 932  
    error handling, 933–935  
    examples of, 926–928  
    exceptions, 932  
    fault tolerance, 930  
    fragmentation, 936, 937  
    free-store, 936–940  
    hard real time, 931  
    ideals, 932–933  
    maintenance, 929  
    memory management, 940–942  
    **new** operator, 932  
    predictability, 931, 932  
    real-time constraints, 931  
    real-time response, 928  
    reliability, 928  
    resource leaks, 931

- Embedded systems, *continued*  
     resource limitations, 928  
     soft real time, 931  
     special concerns, 928–929
- Empty  
     **empty()**, is container empty? 1150  
     lists, 729  
     sequences, 729  
     statements, 101
- Empty statement, 1035–1036
- Encapsulation, 505
- Enciphering (Encryption), example, 969–974
- end()**  
     iterator, 1148  
     **string**, 851, 1177  
     **vector**, 722
- End of file  
     **eof()**, 355, 1171  
     file streams, 366  
     I/O error, 355  
     **stringstream**, 395
- End of input, 124
- End of line **\$** (in regular expressions), 873, 1178
- Ending programs. *See* Termination
- endl** manipulator, 1174
- ends** manipulator, 1174
- English grammar *vs.* programming grammar, 193–194
- enum**, 318–321, 1042. *See also* Enumerations
- Enumerations, 318–321, 1107–1108  
     **enum**, 318–321, 1042  
     enumerators, 318–321, 1107–1108
- EOF** macro, 1053–1054
- eof()** stream state, 355, 1171
- equal()**, 759, 1153
- Equal **==**, 67, 1088
- Equality operators, expressions, 1088
- equal\_range()**, 758, 796
- equal\_to()**, 1163
- erase()**  
     **list**, 742–745, 1150  
     list operations, 615–617  
     **string**, 851, 1177  
     **vector**, 745–747
- errno**, error indicator, 918–919, 1182
- error()** example, 142–143  
     passing multiple strings, 152
- Error diagnostics, templates, 683
- Error handling. *See also* Errors; Exceptions  
     % for floating-point numbers, 230–231  
     catching exceptions, 239–241
- files fail to open, 389
- GUIs, 576
- hardware replication, 934
- I/O errors. *See* I/O errors
- I/O streams, 1171  
     mathematical errors, 918–919  
     modular systems, 934–935  
     monitoring subsystems, 935  
     negative numbers, 229–230  
     positioning in files, 393–394  
     predictable errors, 933  
     recovering from errors, 239–241  
     regular expressions, 878–880  
     resource leaks, 934  
     self-checking, 934  
     STL (Standard Template Library), 1137–1138  
     testing for errors, 225–229  
     transient errors, 934
- vector** resource exceptions, 702
- Error messages. *See also* Reporting errors; **error()**  
     example; **runtime\_error**  
     exceptions, printing, 150–151  
     templates, 683  
     writing your own, 142
- Errors, 1219. *See also* Debugging; Testing  
     classifying, 134  
     compile-time, 48–50, 134, 136–137  
     detection ideal, 135  
     **error()**, 142–143  
     estimating results, 157–158  
     incomplete programs, 136  
     input format, 64–65  
     link-time, 134, 139–140  
     logic, 134, 154–156  
     poor specifications, 136  
     recovering from, 239–241. *See also* Exceptions  
     sources of, 136  
     syntax, 137–138  
     translation units, 139–140  
     type mismatch, 138–139  
     undeclared identifier, 258  
     unexpected arguments, 136  
     unexpected input, 136  
     unexpected state, 136
- Errors, run-time, 134, 140–142. *See also*  
     Exceptions  
     callee responsibility, 143–145  
     caller responsibility, 142–143  
     hardware violations, 141  
     reasons for, 144–145  
     reporting, 145–146

- Essential operations, 640–646  
Estimating development resources, 177  
Estimating results, 157–158  
Examples  
    age distribution, 538–539  
    calculator. *See* Calculator example  
    **Date**. *See* **Date** example  
    deciphering, 969–974  
    deleting repeated words, 71–73  
    dictionary, 123–125, 788  
    Dow Jones tracking, 782–785  
    email analysis, 855–865  
    embedded systems, 926–928  
    enciphering (encryption), 969–974  
    exponential function, 527–528  
    finding largest element, 713–716, 723–724  
    fruits, 779–782  
    Gaussian elimination, 910–911  
    graphics, 414–418, 436  
    graphing data, 537–539  
    graphing functions, 527–528  
    GUI (graphical user interface), 565–569,  
        573–574, 576–577  
    Hello, World! 45–46  
    intrusive containers, 1059–1065  
    **Lines\_window**, 565–569, 573–574, 576–577  
    **Link**, 613–622  
    list (doubly linked), 613–622  
    **map** container, 779–785  
    **Matrix**, 908–914  
    palindromes, 659–662  
    **Pool** allocator, 940–941  
    **Punct\_stream**, 401–405  
    reading a single value, 359–363  
    reading a structured file, 367–376  
    regular expressions, 880–885  
    school table, 880–885  
    searching, 864–872  
    sequences, 723–724  
    **Stack** allocator, 942–943  
    TEA (Tiny Encryption Algorithm),  
        969–974  
    text editor, 734–741  
    **vector**. *See* **vector** example  
    **Widget** manipulation, 565–569,  
        1213–1216  
    windows, 565–569  
    word frequency, 777–779  
    writing a program. *See* Calculator example  
    writing files, 352–354  
    ZIP code detection, 864–872
- <exception>**, 1135  
Exceptions, 146–150, 1125–1126. *See also* Error  
    handling; Errors  
    bounds error, 149  
    C++ and C, 1026  
    **catch**, 147, 239–241, 1125–1126  
    **cerr**, 151–152  
    **cout**, 151–152  
    destructors, 1126  
    embedded systems, 932  
    error messages, printing, 150–151  
    exception, 152, 1138–1139  
    failure to catch, 153  
    GUIs, 576  
    input, 150–153  
    **narrow\_cast** example, 153  
    off-by-one error, 149  
    **out\_of\_range**, 149–150, 152  
    overview, 146–147  
    RAII (Resource Acquisition Is Initialization),  
        1125  
    range errors, 148–150  
    re-throwing, 702, 1126  
    **runtime\_error**, 142, 151, 153  
    stack unwinding, 1126  
    standard library exceptions, 1138–1139  
    terminating a program, 142  
    **throw**, 147, 1125  
    truncation, 153  
    type conversion, 153  
    uncaught exception, 153  
    user-defined types, 1126  
    **vector** range checking, 693–694  
    **vector** resources. *See* **vector**  
Executable code, 48, 1219  
Executing a program, 11, 1200–1201  
**exit()**, terminating a program, 1194–1195  
**explicit** constructor, 642–643, 1038  
Expression, 94–95, 1086–1090  
    coding standards, 980–981  
    constant expressions, 1093  
    conversions, 1091–1093  
    debugging, 161  
    grouping (**0**), 95, 867, 873, 876  
    lvalue, 94–95, 1090  
    magic constants, 96, 143, 232–234, 723  
    memory management, 1094–1095  
    mixing types, 99  
    non-obvious literals, 96  
    operator precedence, 95  
    operators, 97–99, 1086–1095

- Expression, *continued*  
     order of operations, 181  
     precedence, 1090  
     preserving values, 1091  
     promotions, 99, 1091  
     rvalue, 94–95, 1090  
     scope resolution, 1086  
     type conversion, 99–100, 1095  
     usual arithmetic conversions, 1092
- Expression statement, 100
- extern**, 259, 1033
- Extracting text from files, 856–861, 864–865
- ## F
- f/F** suffix, 1079
- fail()** stream state, 355, 1171
- Falling through end of functions, 274
- false**, 1038
- Fault tolerance, 930
- fclose()**, 1053–1054, 1186
- Feature creep, 188, 201, 1219
- Feedback, programming, 36
- Fields, formatting, 387–388
- FILE**, 1053–1054
- File I/O, 349–350  
     binary I/O, 391  
     **close()**, 352  
     closing files, 352, 1186  
     converting representations, 374–376  
     modes, 1186  
     **open()**, 352  
     opening files. *See* Opening files  
     positioning in files, 393–394  
     reading. *See* Reading files  
     writing. *See* Writing files
- Files, 1219. *See also* File I/O  
     C++ and C, 1053–1054  
     opening and closing, C-style I/O, 1186
- fill()**, 1157
- fill\_n()**, 1157
- Fill color example, 462–465, 500
- find()**, 758–761  
     associative container operations, 1151  
     finding links, 615–617  
     generic use, 761–763  
     nonmodifying sequence algorithms, 1153  
     string operations, 851, 1177
- find\_end()**, 1153
- find\_first\_of()**, 1153
- find\_if()**, 758, 763–764
- Finding. *See also* Matching; Searching  
     associative container operations, 1151  
     elements, 758  
     links, 615–617  
     patterns, 864–865, 869–872  
     strings, 851, 1177
- fixed** format, 387
- fixed** manipulator, 385, 1174
- <float.h>**, 894, 1181
- Floating-point, 63, 891, 1219  
     % remainder (modulo), 201  
     assigning integers to, 892–893  
     assigning to integers, 893  
     conversions, 1092  
     **fixed** format, 387  
     **general** format, 387  
     input, 182, 201–202  
     integral conversions, 1091–1092  
     literals, 182, 1079  
     mantissa, 893  
     output, formatting, 384–385  
     precision, 386–387  
     and real numbers, 891  
     rounding, 386  
     **scientific** format, 387  
     truncation, 893  
     **vector** example, 120–123
- float** type, 1099
- floor()**, 917, 1181
- FLTK (Fast Light Toolkit), 418, 1204  
     code portability, 418  
     color, 451, 465–467  
     current style, obtaining, 500  
     downloading, 1204  
     fill, 465  
     in graphics code, 436  
     installing, 1205  
     lines, drawing, 454, 458  
     outlines, 465  
     rectangles, drawing, 465  
     testing, 1206  
     in Visual Studio, 1205–1206  
     waiting for user action, 559–560, 569–570
- flush** manipulator, 1174
- Flushing a buffer, 240–241
- Fonts for Graphics example, 468–470
- fopen()**, 1053–1054, 1186
- for**-statement, 111–113  
     vs. **while**, 122

**for\_each()**, 119, 1153  
Ford, Henry, 806  
Formal arguments. *See* Parameters  
Formatting. *See also* C-style I/O; I/O streams;  
    Manipulators  
        *See also* C-style I/O, 1050–1054  
        *See also* I/O streams, 1172–1173  
    case, 397–398  
        *See also* Manipulators, 1173–1175  
    fields, 387–388  
    precision, 386–387  
    whitespace, 397  
Fortran language, 821–823  
    array indexing, 899  
    **complex**, 920  
    subscripting, 899  
Forward declarations, 261  
Forward iterators, 752, 1142  
**fprintf()**, 1051–1052, 1187  
Fragmentation, embedded systems, 936, 937  
**free()**, deallocate, 1043–1044, 1193  
Free store (heap storage)  
    allocation, 593–594  
    C++ and C, 1043–1045  
    deallocation, 598–600  
    **delete**, 598–600, 601–605  
    and destructors. *See* Destructors  
    embedded systems, 936–940  
    garbage collection, 600  
    leaks, 598–600, 601–605  
    **new**, 593–594  
    object lifetime, 1085  
Freeing memory. *See* Deallocating memory  
**friend**, 1038, 1111  
**from\_string()** example, 853–854  
**front()**, first element, 1149  
**front\_inserter()**, 1162  
**fstream()**, 1170  
**<fstream>**, 1134  
**fstream** type, 350–352  
Fully qualified names, 295–297  
**Function** example, 443, 525–528  
Function , 47, 113–117. *See also* Member functions  
    accessing class members, 1111  
    arguments. *See* Function arguments  
    in base classes, 504  
    body, 47, 114  
    C++ and C, 1028–1032  
    callback, GUIs, 556–559  
    calling, 1103  
    cleaning up, 234–235  
    coding standards, 980–981  
    common style, 490–491  
    debugging, 160  
    declarations, 117, 1103  
    definition, 113–115, 272, 1219  
    in derived classes, 501, 505  
    falling through, 274  
    formal arguments. *See* Function parameter  
        (formal argument)  
    **friend** declaration, 1111  
    generic code, 491  
    global variables, modifying, 269  
    graphing. *See* **Function** example  
    inline, 316, 1026  
    linkage specifications, 1106  
    naming. *See* Namespaces  
    nesting, 270  
    organizing. *See* Namespaces  
    overloading, 321–323, 526, 1026  
    overload resolution, 1104–1105  
    parameter, 115. *See also* Function parameter  
        (formal argument)  
    pointer to, 1034–1036  
    post-conditions, 165–166  
    pre-conditions, 163–165  
    pure virtual, 1221  
    requirements, 153. *See also* Pre-conditions  
    **return**, 113–115, 272–273, 1103  
    return type, 47, 272–273  
    standard mathematical, 528, 1181–1182  
    types as parameters. *See* Template  
    uses for, 115–116  
**virtual**, 1034–1036. *See also* Virtual functions  
Function activation record, 287  
Function argument. *See also* Function parameter  
    (formal argument); Parameters  
    checking, 284–285  
    conversion, 284–285  
    declaring, 272–273  
    formal. *See* Parameters  
    naming, 273  
    omitting, 273  
    passing. *See* Function call  
Function call, 285  
call stack, 290  
**expression()** call example, 287–290  
function activation record, 287  
history of, 820  
memory for, 591–592

- Function call, *continued*
- `()` operator, 766
  - pass by `const` reference, 276–278, 281–284
  - pass by non-`const` reference, 281–284
  - pass by reference, 279–284
  - pass by value, 276, 281–284
  - recursive, 289
  - stack growth, 287–290. *See also* Function
    - activation record
    - temporary objects, 282
  - Function-like macros, 1056–1058
  - Function member
    - definition, 305–306
    - same name as class. *See* Constructors
  - Function objects, 765–767
    - `()` function call operator, 766
    - abstract view, 766–767
    - adaptors, 1164
    - arithmetic operations, 1164
    - parameterization, 767
    - predicates, 767–768, 1163
  - Function parameter (formal argument)
    - ... ellipsis, unchecked arguments, 1105–1106
    - pass by `const` reference, 276–278, 281–284
    - pass by non-`const` reference, 281–284
    - pass by reference, 279–284
    - pass by value, 276, 281–284
    - temporary objects, 282
    - unused, 272
  - Function template
    - algorithms, 682–683
    - argument deduction, 689–690
    - parameterized functions, 682–683
  - `<functional>`, 1133, 1163
  - Functional cast, 1095
  - Functional programming, 823
  - Fused multiply-add, 904
- ## G
- Gadgets. *See* Embedded systems
- Garbage collection, 600, 938–939
- Gaussian elimination, 910–911
- `gcount()`, 1172
- `general` format, 387
- `general` manipulator, 385
- `generate()`, 1157
- `generate_n()`, 1157
- Generic code, 491
- Generic programming, 682–683, 816, 1219
- Geometric shapes, 427
- `get()`, 1172
- `getc()`, 1052, 1191
- `getchar()`, 1053, 1191
- `getline()`, 395–396, 851, 855, 1172
- `gets()`, 1052
- C++ alternative `>>`, 1053
  - dangerous, 1052
  - `scanf()`, 1190
- `get_token()` example, 196
- GIF images, 480–482
- Global scope, 267, 270, 1082
- Global variables
  - functions modifying, 269
  - memory for, 591–592
  - order of initialization, 292–294
- Going out of scope, 268–269, 291
- `good()` stream state, 355, 1171
- GP. *See* Generic programming
- Grammar example
  - alternation, patterns, 194
  - English grammar, 193–194
  - `Expression` example, 197–200, 202–203
  - parsing, 190–193
  - repetition, patterns, 194
  - rules *vs.* tokens, 194
  - sequencing rules, 195
  - terminals. *See* Tokens
  - writing, 189, 194–195
- Graph example. *See also* Grids, drawing
  - `Axis`, 424–426
  - coordinates, 426–427
  - drawing, 426–427
  - points, labeling, 474–476
- `Graph.h`, 421–422
- Graphical user interfaces. *See* GUIs (graphical user interfaces)
- Graphics, 412. *See also* Graphics example; `Color` example; `Shape` example
  - displaying, 479–482
  - display model, 413–414
  - drawing on screen, 423–424
  - encoding, 480
  - filling shapes, 431
  - formats, 480
  - geometric shapes, 427
  - GIF, 480–482
  - graphics libraries, 481–482
  - graphs, 426–427
  - images from files, 433–434
  - importance of, 412–413
  - JPEG, 480–482

- line style, 431  
loading from files, 433–434  
screen coordinates, 419–420  
selecting a sub-picture from, 480  
user interface. *See* GUIs (graphical user interfaces)
- Graphics example  
**Graph.h**, 421–422  
GUI system, giving control to, 423  
header files, 421–422  
**main()**, 421–422  
**Point.h**, 444  
points, 426–427  
**Simple\_window.h**, 444  
**wait\_for\_button()**, 423  
**Window.h**, 444
- Graphics example, design principles  
access control. *See* Access control  
**attach()** *vs.* **add()**, 491–492  
class diagram, 505  
class size, 489–490  
common style, 490–491  
data modification access, 492–493  
generic code, 491  
inheritance, interface, 513–514  
inheritances, implementation, 513–514  
mutability, 492–493  
naming, 491–492  
object-oriented programming, benefits of, 513–514  
operations, 490–491  
private data members, 492–493  
protected data, 492–493  
public data, 492–493  
types, 488–490  
width/height, specifying, 490
- Graphics example, GUI classes, 442–444. *See also* Graphics example, interfaces  
**Button**, 443  
**In\_box**, 443  
**Menu**, 443  
**Out\_box**, 443  
**Simple\_window**, 422–424, 443  
**Widget**, 561–563, 1209–1210  
**Window**, 443, 1210–1212
- Graphics example, interfaces, 442–443. *See also* Graphics example, GUI classes  
**Axis**, 424–426, 443, 529–532  
**Circle**, 469–472, 497  
**Closed\_polyline**, 456–458  
**Color**, 450
- Ellipse**, 472–474  
**Function**, 443, 524–528  
**Image**, 443, 479–482  
**Line**, 445–448  
**Line\_style**, 452–455  
**Lines**, 448–450, 497  
**Mark**, 478–479  
**Marked\_polyline**, 474–476  
**Marks**, 476–477, 497  
**Open\_polyline**, 455–456, 497  
**Point**, 426–427, 445  
**Polygon**, 427–428, 458–460, 497  
**Rectangle**, 428–431, 460–465, 497  
**Shape**, 444–445, 449, 493–494, 513–514  
**Text**, 431–433, 467–470
- Graphing data example, 538–546  
Graphing functions example, 520–524, 532–537
- Graph\_lib** namespace, 421–422  
**greater()**, 1163  
Greater than **>**, 67, 1088  
Greater than or equal **>=**, 1088  
**greater\_equal()**, 1163  
Green marginal alerts, 3  
Grids, drawing, 448–449, 452–455  
Grouping regular expressions, 867, 873, 876  
Guarantees, 701–702  
Guidelines. *See* Ideals  
GUIs (graphical user interfaces), 552–553. *See also* Graphics example, GUI classes  
callback functions, 556–559  
callback implementation, 1208–1209  
**cb\_next()** example, 556–559  
common problems, 575–577  
control inversion, 569–570  
controls. *See* Widget example  
coordinates, computer screens, 419–420  
debugging, 575–577  
error handling, 576  
examples, 565–569, 573–574, 576–577  
exceptions, 576  
FLTK (Fast Light Toolkit), 418  
layers of code, 557  
**next()** example, 558–559  
pixels, 419–420  
portability, 418  
standard library, 418–419  
toolkit, 418  
**vector\_ref** example, 1212–1213  
**vector** of references, simulating, 1212–1213

GUIs (graphical user interfaces), *continued*  
 wait loops, 559–560  
**wait\_for\_button()** example, 559–560  
 waiting for user action, 559–560,  
 569–570  
**Widget** example, 561–569, 1209–1210,  
 1213–1216  
**Window** example, 565–569, 1210–1212  
 GUI system, giving control to, 423

## H

.h file suffix, 46  
 Half open sequences, 119, 721  
 Hard real-time, 931, 981–982  
 Hardware replication, error handling, 934  
 Hardware violations, 141  
 Hashed container. *See* **unordered\_map**  
 Hash function, 785–786  
 Hashing, 785  
 Hash tables, 785  
 Hash values, 785  
 Header files, 46, 1219  
   C standard library, 1135–1136  
   declarations, managing, 264  
   definitions, managing, 264  
   graphics example, 421–422  
   including in source files, 264–266, 1129  
   multiple inclusion, 1059  
   standard library, 1133–1134  
 Headers. *See* Header files  
 Heap algorithm, 1160  
 Heap memory, 592, 935–936, 1084, 1160. *See also*  
   Free store (heap storage)  
 Hejlsberg, Anders, 831  
 “Hello, World!” program, 45–47  
 Helper functions  
   == equality, 333  
   != inequality, 333  
   class interfaces, 332–334  
   **Date** example, 309–310, 332–333  
   namespaces, 333  
   validity checking date values, 310  
**hex** manipulator, 382–383, 1174  
 Hexadecimal digits, 397  
 Hexadecimal number system, 381–383,  
 1077–1078  
 Hiding information, 1220  
 Hopper, Grace Murray, 824–825  
 Hyperbolic cosine, **cosh()**, 918

Hyperbolic sine, **sinh()**, 918, 1182  
 Hyperbolic tangent, **tanh()**, 917

## I

I/O errors  
**bad()** stream state, 355  
**clear()**, 355–358  
 end of file, 355  
**eof()** stream state, 355  
 error handling, 1171  
**fail()** stream state, 355  
**good()** stream state, 355  
**ios\_base**, 357  
 recovering from, 355–358  
 stream states, 355  
 unexpected errors, 355  
**unget()**, 355–358  
 I/O streams, 1168–1169  
**>>** input operator, 855  
**<<** output operator, 855  
**cerr**, standard error output stream, 151–152,  
 1169, 1189  
**cin** standard input, 347  
 class hierarchy, 855, 1170–1171  
**cout** standard output, 347  
 error handling, 1171  
 formatting, 1172–1173  
**fstream**, 388–390, 393, 1170  
**get()**, 855  
**getline()**, 855  
 header files, 1134  
**ifstream**, 388–390, 1170  
 input operations, 1172  
 input streams, 347–349  
 iostream library, 347–349, 1168–1169  
**istream**, 347–349, 1169–1170  
**istringstream**, 1170  
**ofstream**, 388–390, 1170  
**ostream**, 347–349, 1168–1169  
**ostringstream**, 388–390, 1170  
 output operations, 1173  
 output streams, 347–349  
 standard manipulators, 382, 1173–1174  
 standard streams, 1169  
 states, 1171  
 stream behavior, changing, 382  
 stream buffers, **streambufs**, 1169  
 stream modes, 1170  
**string**, 855

**stringstream**, 395, 1170  
throwing exceptions, 1171  
unformatted input, 1172  
**IBM**, 823  
**Ichbiah**, Jean, 832  
**IDE** (interactive development environment), 52  
**Ideals**  
abstraction level, 812–813  
bottom-up approach, 811  
class interfaces, 323  
code structure, 810–811  
coding standards, 976–977  
consistency, 814–815  
correct approaches, 811  
correctness, 810  
definition, 1219  
direct expression of ideas, 811–812  
efficiency, 810  
embedded systems, 932–933  
importance of, 8  
**KISS**, 815  
maintainability, 810  
minimalism, 814–815  
modularity, 813–814  
overview, 808–809  
performance, 810  
software, 34–37  
on-time delivery, 810  
top-down approach, 811  
**Identifiers**, 1081. *See also* Names  
reserved, 75–76. *See also* Keywords  
**if**-statements, 102–104  
**#ifdef**, 1058–1059  
**#ifndef**, 1058–1059  
**ifstream** type, 350–352  
**imag()**, imaginary part, 920, 1183  
**Image** example, 443, 479–482  
**Images**. *See* Graphics  
Imaginary part, 920  
Immutable values, class interfaces, 330–332  
Implementation, 1219  
class, 306–308  
inheritance, 513–514  
programs, 36  
Implementation-defined feature, 1075  
Implicit conversions, 642–643  
**in** mode, 389, 1170  
**In\_box** example, 443, 563–564  
In-class member definition, 1112  
**#include**, 46, 264–266, 1128–1129  
Include guard, 1059  
**includes()**, 1159  
Including headers, 1129. *See also* **#include**  
Incrementing **++**, 66, 721  
iterators, 721, 750, 1140–1141  
pointers, 651–652  
variables, 73–74, 97–98  
Indenting nested code, 271  
Inequality **!=** (not equal), 67, 1088, 1101  
**complex**, 919, 1183  
containers, 1151  
helper function, 333  
iterators, 721, 1141  
**string**, 67, 851, 1176  
Infinite loop, 1219  
Infinite recursion, 198, 1220  
Information hiding, 1220  
Information processing, 32  
Inheritance  
class diagram, 505  
definition, 504  
derived classes, 1116–1117  
embedded systems, 951–954  
history of, 834  
implementation, 513–514  
interface, 513–514  
multiple, 1117  
pointers *vs.* references, 612–613  
templates, 686–687  
Initialization, 69–73, 1220  
**{} initialization notation**, 83  
arrays, 596–598, 654–656  
constants, 262, 329–330, 1099  
constructors, 310–312  
**Date** example, 309–312  
default, 263, 327, 1085  
invariants, 313–314, 701–702  
menus, 571  
pointers, 596–598, 657  
pointer targets, 596–598  
**Token** example, 184  
**initializer\_list**, 630  
**inline**, 1037  
Inline  
functions, 1026  
member functions, 316  
**inner\_product()**, 759. *See also* Dot product  
description, 774–775  
generalizing, 775–776

**inner\_product()**, *continued*  
 matrices, 904  
 multiplying sequences, 1184  
 standard library, 759, 770  
**inplace\_merge()**, 1158  
 Input, 60–62. *See also* Input **>>**; I/O streams  
 binary I/O, 390–393  
 C++ and C, 1052–1053  
 calculator example, 179, 182, 185, 201–202,  
   206–208  
 case sensitivity, 64  
**cin**, standard input stream, 61  
 dividing functions logically, 359–362  
 files. *See* File I/O  
 format errors, 64–65  
 individual characters, 396–398  
 integers, 383–384  
**istringstream**, 394  
 line-oriented input, 395–396  
 newline character **\n**, 61–62, 64  
 potential problems, 358–363  
 prompting for, 61, 179  
 separating dialog from function, 362–363  
 a series of values, 356–358  
 a single value, 358–363  
 source of exceptions, 150–153  
**stringstream**, 395  
 tab character **\t**, 64  
 terminating, 61–62  
 type sensitivity, 64–65  
 whitespace, 64  
**Input >>**, 61  
 case sensitivity, 64  
**complex**, 920, 1183  
 formatted input, 1172  
 multiple values per statement, 65  
 strings, 851, 1177  
 text input, 851, 855  
 user-defined, 365  
 whitespace, ignoring, 64  
 Input devices, 346–347  
 Input iterators, 752, 1142  
 Input loops, 365–367  
 Input/output, 347–349. *See also* Input; Output  
 buffering, 348, 406  
 C++ and C. *See* stdio  
 computation overview, 91  
 device drivers, 346  
 errors. *See* I/O errors  
 files. *See* File I/O  
 formatting. *See* Manipulators; **printf()**

irregularity, 380  
**istream**, 347–354  
 natural language differences, 406  
**ostream**, 347–354  
 regularity, 380  
 streams. *See* I/O streams  
 strings, 855  
 text in GUIs, 563–564  
 whitespace, 397, 398–405  
**Input prompt >**, 223  
 Inputs, testing, 1001  
 Input streams, 347–349. *See also* I/O streams  
**insert()**  
**list**, 615–617, 742–745  
**map** container, 782  
**string**, 851, 1150, 1177  
**vector**, 745–747  
**inserter()**, 1162  
 Inserters, 1162–1163  
 Inserting  
**list** elements, 742–745  
 into **strings**, 851, 1150, 1177  
**vector** elements, 745–747  
 Installing  
 FLTK (Fast Light Toolkit), 1205  
 Visual Studio, 1198  
 Instantiation, templates, 681, 1123–1124  
**int**, integer type, 66–67, 78, 1099  
 bits in memory, 78, 955  
 Integers, 77–78, 890–891, 1220  
 assigning floating-point numbers to, 893  
 assigning to floating-point numbers, 892–893  
 decimal, 381–383  
 input, formatting, 383–384  
 largest, finding, 917  
 literals, 1077  
 number bases, 381–383  
 octal, 381–383  
 output, formatting, 381–383  
 reading, 383–384  
 smallest, finding, 917  
 Integral conversions, 1091–1092  
 Integral promotion, 1091  
 Interactive development environment (IDE), 52  
 Interface classes. *See* Graphics example, interfaces  
 Interfaces, 1220  
 classes. *See* Class interfaces  
 inheritance, 513–514  
 user. *See* User interfaces  
**internal** manipulator, 1174  
 Intrusive containers, example, 1059–1065

- Invariants, 313–314, 1220. *See also* Post-conditions;
- Pre-conditions
  - assertions, 163
  - Date** example, 313–314
  - debugging, 162–163
  - default constructors, 641
  - documenting, 815
  - invention of, 828
  - Polygon** example, 460
- Invisible. *See* Transparency
- <iomanip>**, 1134, 1173
- <ios>**, 1134, 1173
- <iostfwd>**, 1134
- iostream**
- buffers, 406
  - C++ and C, 1050
  - exceptions, 1138
  - library, 347–349
- <iostream>**, 1134, 1173
- Irregularity, 380
- is\_open()**, 1170
- isalnum()** classify character, 397, 1175
- isalpha()** classify character, 247, 397, 1175
- iscntrl()** classify character, 397, 1175
- isdigit()** classify character, 397, 1175
- isgraph()** classify character, 397, 1175
- islower()** classify character, 397, 1175
- isprint()** classify character, 397, 1175
- ispunct()** classify character, 397, 1175
- isspace()** classify character, 397, 1175
- istream**, 347–349, 1169–1170
- >>**, text input, 851, 1172
  - >>**, user-defined, 365
  - binary I/O, 390–393
  - connecting to input device, 1170
  - file I/O, **fstream**, 349–354, 1170
  - get()**, get a single character, 397
  - getline()**, 395–396, 1172
  - stringstreams**, 395
  - unformatted input, 395–396, 1172
  - using together with stdio, 1050
- <istream>**, 1134, 1168–1169, 1173
- istream\_iterator** type, 790–793
- istringstream**, 394
- isupper()** classify character, 397, 1175
- isxdigit()** classify character, 397, 1175
- Iteration. *See also* Iterators
- control variables, 110
  - definition, 1220
  - example, 737–741
  - linked lists, 727–729, 737–741
- loop variables, 110–111
- for**-statements, 111–113
- strings, 851
- through values. *See* **vector**
- while**-statements, 109–111
- iterator**, 1147
- <iterator>**, 1133, 1162
- Iterators, 721–722, 1139–1140, 1220. *See also* STL
- iterators
  - bidirectional iterator, 752
  - category, 752, 1142–1143
  - containers, 1143–1145, 1148
  - empty list, 729
  - example, 737–741
  - forward iterator, 752
  - header files, 1133–1134
  - input iterator, 752
  - operations, 721, 1141–1142
  - output iterator, 752
  - vs.* pointers, 1140
  - random-access iterator, 752
  - sequence of elements, 1140–1141
- iter\_swap()**, 1157
- J**
- Japanese age distribution example, 538–539
- JPEG images, 480–482
- K**
- Kernighan, Brian, 838–839, 1022–1023
- key\_comp()**, 1152
- key\_compare**, 1147
- key\_type**, 1147
- Key, value pairs, containers for, 776
- Keywords, 1037–1038, 1081–1082
- KISS, 815
- Knuth, Don, 808
- K&R, 838, 1022
- L**
- I/L** suffix, 1077
- \l**, “lowercase character,” **regex**, 873, 1179
- \L**, “not lowercase character,” **regex**, 874, 1179
- Label
- access control, 306, 511
  - case**, 106–108
  - graph example, 529–532
  - of statement, 1096

Lambda expression, 560–561  
 Largest integer, finding, 917  
 Laws of optimization, 931  
 Layers of code, GUIs, 557  
 Layout rules, 979, 1034  
 Leaks, memory, 598–600, 601–605, 937  
 Leap year, 309  
**left** manipulator, 1174  
 Legal programs, 1075  
**length()**, 851, 1176  
 Length of strings, finding, 851, 1046, 1176  
**less()**, 1163  
 Less than **<**, 1088  
 Less than or equal **<=**, 67, 1088  
**less\_equal()**, 1163  
 Letters, identifying, 247, 397  
**lexical\_cast**, 855  
 Lexicographical comparison  
     **<=** comparison, 1176  
     **<** comparison, 1176  
     **>=** comparison, 1176  
     **>** comparison, 1176  
     **<** comparison, 851  
 C-style strings, 1046  
**lexicographical\_compare()**, 1162  
 Libraries, 51, 1220. *See also* Standard library  
     role in debugging, 160  
     uses for, 177  
 Lifetime, objects, 1085–1086, 1220  
 Limit macros, 1181  
**<limits>**, 894, 1135, 1180  
 Limits, 894–895  
**<limits.h>**, 894, 1181  
 Linear equations example, 908–914  
     **back\_substitution()**, 910–911  
     **classic\_elimination()**, 910–911  
     Gaussian elimination, 910–911  
     pivoting, 911–912  
     testing, 912–914  
 Line comment **//**, 45  
**Line** example, 445–447  
     *vs.* **Lines**, 448  
 Line-oriented input, 395–396  
**Lines** example, 448–450, 497  
     *vs.* **Line**, 448  
 Lines (graphic), drawing. *See also* Graphics;  
     **draw\_lines()**  
     on graphs, 529–532  
     line styles, 452–455  
     multiple lines, 448–450  
     single lines, 445–447  
     styles, 431, 454  
     visibility, 500  
 Lines (of text), identifying, 736–737  
**Line\_style** example, 452–455  
**Lines\_window** example, 565–569, 573–574, 576–577  
**Link** example, 613–622  
 Link-time errors. *See* Errors, link-time  
 Linkage convention, C, 1033  
 Linkage specifications, 1106  
 Linked lists, 725. *See also* Lists  
 Linkers, 51, 1220  
 Linking programs, 51  
 Links, 613–615, 620–622, 725  
 Lint, consistency checking program, 836  
 Lisp language, 825–826  
**list**, 727, 1146–1151  
     {} initialization notation, 83  
     **add()**, 615–617  
     **advance()**, 615–617  
     **back()**, 737  
     **erase()**, 615–617, 742–745  
     **find()**, 615–617  
     **insert()**, 615–617, 742–745  
     operations, 615–617  
     properties, 741–742  
     referencing last element, 737  
     sequence containers, 1144  
     subscripting, 727  
**<list>**, 1133  
 Lists  
     containers, 1150  
     doubly linked, 613, 725  
     empty, 729  
     erasing elements, 742–745  
     examples, 613–615, 734–741  
     finding links, 615–617  
     getting the *n*th element, 615–617  
     inserting elements, 615–617, 742–745  
     iteration, 727–729, 737–741  
     link manipulation, 615–617  
     links, examples, 613–615, 620–622, 726  
     operations, 726–727  
     removing elements, 615–617  
     singly linked, 612–613, 725  
     **this** pointer, 618–620  
 Literals, 62, 1077, 1220  
     character, 161, 1079–1080  
     decimal integer, 1077  
     in expressions, 96  
     **f/F** suffix, 1079  
     floating-point, 1079

hexadecimal integer, 1077  
 integer, 1077  
**I/L** suffix, 1077  
 magic constants, 96, 143, 232–234, 723  
 non-obvious, 96  
 null pointer, **0**, 1081  
 number systems, 1077–1079  
 octal integer, 1077  
 special characters, 1079–1080  
 string, 161, 1080  
 termination, debugging, 161  
 for types, 63  
**u/U** suffix, 1077  
 unsigned, 1077  
 Local (automatic) objects, lifetime, 1085  
 Local classes, nesting, 270  
 Local functions, nesting, 270  
 Local scope, 267, 1083  
 Local variables, array pointers, 658  
 Locale, 406  
**<locale>**, 1135  
**log()**, 918, 1182  
**log10()**, 918, 1182  
 Logic errors. *See* Errors, logic  
 Logical and **&&**, 1089, 1094  
 Logical operations, 1094  
 Logical or **||**, 1089, 1094  
**logical\_and()**, 1163  
**logical\_not()**, 1163  
**logical\_or()**, 1163  
 Logs, graphing, 528  
**long** integer, 955, 1099  
 Look-ahead problem, 204–209  
 Loop, 110–111, 112, 1220  
     examples, parser, 200  
     infinite, 198, 1219  
     testing, 1005–1006  
     variable, 110–111, 112  
 Lovelace, Augusta Ada, 832  
**lower**, 878, 1179  
**lower\_bound()**, 796, 1152, 1158  
 Lower case. *See* Case (of characters)  
 Lucent Bell Labs, 838  
 Lvalue, 94–95, 1090

**M**

Machine code. *See* Executable code  
 Macros, 1055–1056  
     conditional compilation, 1058–1059  
**#define**, 1056–1058, 1129  
     function-like, 1056–1058  
**#ifdef**, 1058–1059  
**#ifndef**, 1059  
**#include**, 1058, 1128–1129  
 include guard, 1059  
 naming conventions, 1055  
 syntax, 1058  
 uses for, 1056  
 Macro substitution, 1129  
 Maddock, John, 865  
 Magic constants, 96, 143, 232–234, 723  
 Magical approach to programming, 10  
**main()**, 46–47  
     arguments to, 1076  
     global objects, 1076  
     return values, 47, 1075–1076  
     starting a program, 1075–1076  
 Maintainability, software, 35, 810  
 Maintenance, 929  
**make\_heap()**, 1160  
**make\_pair()**, 782, 1165–1166  
**make\_unique()**, 1167  
**make\_vec()**, 702  
**malloc()**, 1043–1044, 1193  
 Manipulators, 382, 1173–1174  
     complete list of, 1173–1174  
**dec**, 1174  
**endl**, 1174  
**fixed**, 1174  
**hex**, 1174  
**noskipws**, 1174  
**oct**, 1174  
**resetiosflags()**, 1174  
**scientific**, 1174  
**setiosflags()**, 1174  
**setprecision()**, 1174  
**skipws**, 1174  
 Mantissa, 893  
**map**, associative array, 776–782. *See also* **set**;  
     **unordered\_map**  
**I**, subscripting, 777, 1151  
 balanced trees, 780–782  
 binary search trees, 779  
 case sensitivity, **No\_case** example, 795  
 counting words example, 777–779  
 Dow Jones example, 782–785  
 email example, 855–872  
**erase()**, 781, 1150  
 finding elements in, 776–777, 781,  
     1151–1152  
 fruits example, 779–782

**map**, associative array, *continued*  
**insert()**, 782, 1150  
 iterators, 1144  
 key storage, 776  
**make\_pair()**, 782  
**No\_case** example, 782, 795  
**Node** example, 779–782  
 red-black trees, 779  
*vs. set*, 788  
 standard library, 1146–1152  
 tree structure, 779–782  
 without values. *See set*  
**<map>**, 776, 1133  
**mapped\_type**, 1147  
 Marginal alerts, 3  
**Mark** example, 478–479  
**Marked\_polyline** example, 474–476  
**Marks** example, 476–477, 497  
 Mars Rover, 33  
 Matching. *See also* Finding; Searching  
   regular expressions, **regex**, 1177–1179  
   text patterns. *See* Regular expressions  
 Math functions, 528, 1181–1182  
 Mathematics. *See* Numerics  
 Mathematical functions, standard  
   **abs()**, absolute value, 917  
   **acos()**, arccosine, 917  
   **asin()**, arcsine, 918  
   **atan()**, arctangent, 918  
   **ceil()**, 917  
   **<cmath>**, 918, 1135  
   **<complex>**, 919–920  
   **cos()**, cosine, 917  
   **cosh()**, hyperbolic cosine, 918  
   **errno**, error indicator, 918–919  
   error handling, 918–919  
   **exp()**, natural exponent, 918  
   **floor()**, 917  
   **log()**, natural logarithm, 918  
   **log10()**, base-10 logarithm, 918  
   **sin()**, sine, 917  
   **sinh()**, hyperbolic sine, 918  
   **sqrt()**, square root, 917  
   **tan()**, tangent, 917  
   **tanh()**, hyperbolic tangent, 917  
 Matrices, 899–901, 905–906  
**Matrix** library example, 899–901, 905  
**I**, subscripting (C style), 897, 899  
**O**, subscripting (Fortran style), 899  
 accessing array elements, 899–901  
**apply()**, 903  
 broadcast functions, 903  
**clear\_row**, 906  
 columns, 900–901, 906  
 dimensions, 898–901  
 dot product, 904  
 fused multiply-add, 904  
 initializing, 906  
**inner\_product**, 904  
 input/output, 907  
 linear equations example, 910–914  
 multidimensional matrices, 898–908  
 rows, 900–901, 906  
**scale\_and\_add()**, 904  
**slice()**, 901–902, 905  
**start\_row**, 906  
 subscripting, 899–901, 905  
**swap\_columns()**, 906  
**swap\_rows()**, 906  
**max()**, 1161  
**max\_element()**, 1162  
**max\_size()**, 1151  
 McCarthy, John, 825–826  
 McIlroy, Doug, 837, 1032  
 Medicine, computer use, 30  
 Member, 305–307. *See also* Class  
   allocated at same address, 1121  
   class, nesting, 270  
   in-class definition, 1112  
   definition, 1108  
   definitions, 1112  
   out-of-class definition, 1112  
 Member access. *See also* Access control  
   **.** (dot), 1109  
   **::** scope resolution, 315, 1109  
   notation, 184  
   operators, 608  
   **this** pointer, 1110  
   by unqualified name, 1110  
 Member function. *See also* Class members;  
   Constructors; Destructors; **Date** example  
   calls, 120  
   nesting, 270  
   **Token** example, 184  
 Member initializer list, 184  
 Member selection, expressions, 1087  
 Member types  
   containers, 1147  
   templates, 1124  
**memchr()**, 1193

**memcmp()**, 1192  
**memcpy()**, 1192  
**mem\_fn()** adaptor, 1164  
**memmove()**, 1192  
Memory, 588–590  
    addresses, 588  
    allocating. *See* Allocating memory  
    automatic storage, 591–592  
    **bad\_alloc** exception, 1094  
    for code, 591–592  
    C standard library functions, 1192–1193  
    deallocating, 598–600  
    embedded systems, 940–942  
    exhausting, 1094  
    freeing. *See* Deallocation memory  
    free store, 592–594  
    for function calls, 591–592  
    for global variables, 591–592  
    heap. *See* Free store (heap storage)  
    layout, 591–592  
    object layout, 506–507  
    object size, getting, 590–591  
    pointers to, 588–590  
    **sizeof**, 590–591  
    stack storage, 591–592  
    static storage, 591–592  
    text storage, 591–592  
**<memory>**, 1134  
**memset()**, 1193  
**Menu** example, 443, 564–565, 570–575  
**merge()**, 758, 1158  
Messages to the user, 564  
**min()**, 1161  
**min\_element()**, 1162  
Minimalism, ideals, 814–815  
**minus()**, 1164  
Missing copies, 645  
MTT, 825–826, 838  
Modifying sequence algorithms, 1154–1156  
Modularity, ideals, 813–814  
Modular systems, error handling, 934–935  
Modulo (remainder) **%**, 66. *See also* Remainder  
**modulus()**, 1164  
Monitoring subsystems, error handling, 935  
**move()**, 502, 562  
Move assignments, 637–640  
Move backward **-=**, 1101  
Move forward **+=**, 1101  
Move constructors, 637–640  
Moving, 637–640  
Multi-paradigm programming languages, 818  
Multidimensional matrices, 898–908  
**multimap**, 776, 860–861, 1144  
**<multimap>**, 776  
Multiplicative operators, expressions, 1088  
**multiples()**, 1164  
Multiply **\***, 66, 1088  
Multiply and assign **\*=**, 67  
**multiset**, 776, 1144  
**<multiset>**, 776  
Mutability, 492–493, 1220  
    class interfaces, 332–334  
    and copying, 503–504  
**mutable**, 1037  
Mutating sequence algorithms, 1154–1156

## N

**\n** newline, character literal, 61–62, 64, 1079  
Named character classes, in regular expressions, 877–878  
Names, 74–77  
    \_ (underscore), 75, 76  
    capital letters, 76–77  
    case sensitivity, 75  
    confusing, 77  
    conventions, 74–75  
    declarations, 257–258  
    descriptive, 76  
    function, 47  
    length, 76  
    overloaded, 140, 508–509, 1104–1105  
    reserved, 75–76. *See also* Keywords  
**namespace**, 271, 1037  
Namespaces, 294, 1127. *See also* Scope  
    :: scope resolution, 295–296  
    C++ and C, 1042–1043  
    fully qualified names, 295–297  
    helper functions, 333  
    objects, lifetime, 1085  
    scope, 267, 1082  
    **std**, 296–297  
    for the STL, 1136  
    **using** declarations, 296–297  
    **using** directives, 296–297, 1127  
    variables, order of initialization, 292–294  
Naming conventions, 74–77  
    coding standards, 979–980  
    functions, 491–492  
    macros, 1055

Naming conventions, *continued*  
 role in debugging, 160  
 scope, 269  
**narrow\_cast** example, 153  
 Narrowing conversions, 80–83  
 Narrowing errors, 153  
 Natural language differences, 406  
 Natural logarithms, 918  
 Naur, Peter, 827–828  
**negate()**, 1164  
 Negative numbers, 229–230  
 Nested blocks, 271  
 Nested classes, 270  
 Nested functions, 270  
 Nesting  
   blocks within functions, 271  
   classes within classes, 270  
   classes within functions, 270  
   functions within classes, 270  
   functions within functions, 271  
   indenting nested code, 271  
   local classes, 270  
   local functions, 271  
   member classes, 270  
   member functions, 270  
**structs**, 1037  
**new**, 592, 596–598  
 C++ and C, 1026, 1037  
 and **delete**, 1094–1095  
 embedded systems, 932, 936–940  
 example, 593–594  
 exceptions, 1138  
 types, constructing, 1087  
**<new>**, 1135  
 New-style casts, 1040  
**next\_permutation()**, 1161  
 No-throw guarantee, 702  
**noboolalpha**, 1173  
**No\_case** example, 782  
**Node** example, 779–782  
 Non-algorithms, testing, 1001–1008  
 Non-errors, 139  
 Non-intrusive containers, 1059  
 Nonmodifying sequence algorithm, 1153–1154  
 Non-narrowing initialization, 83  
 Nonstandard separators, 398–405  
**norm()**, 919, 1183  
 Norwegian Computing Center, 833–835  
**noshowbase**, 383, 1173  
**noshowpoint**, 1173  
**noshowpos**, 1173  
**noskipws**, 1174  
**not**, synonym for **!** 1037, 1038  
**Not !** 1087  
**not10** adaptor, 1164  
**not20** adaptor, 1164  
 Notches, graphing data example, 529–532, 543–546  
 Not-conforming constructs, 1075  
 Not equal **!=** (inequality), 67, 1088, 1101  
**not\_eq**, synonym for **!=**, 1038  
**not\_equal\_to()**, 1163  
**nouppercase** manipulator, 1174  
**now()**, 1016, 1185  
**nth\_element()**, 1158  
 Null pointer, 598, 656–657, 1081  
**nullptr**, 598  
**Number** example, 189  
 Number systems  
   base-2, binary, 1078–1079  
   base-8, octal, 381–384, 1077–1078  
   base-10, decimal, 381–384, 1077–1078  
   base-16, hexadecimal, 381–384, 1077–1078  
**<numeric>**, 1135, 1183  
 Numerical algorithms. *See* Algorithms, numerical  
 Numerics, 890–891  
 absolute values, 917  
 arithmetic function objects, 1164  
 arrays. *See* **Matrix** library example  
**<cmath>**, 918  
 columns, 895–896  
**complex**, 919–920, 1182–1183  
**<complex>**, 919–920  
 floating-point rounding errors, 892–893  
 header files, 1134  
 integer and floating-point, 892–893  
 integer overflow, 891–893  
 largest integer, finding, 917  
 limit macros, 1181  
 limits, 894  
 mantissa, 893  
 mathematical functions, 917–918  
**Matrix** library example, 897–908  
 multi-dimensional array, 895–897  
**numeric\_limits**, 1180  
 numerical algorithms, 1183–1184  
 overflow, 891–895  
 precision, 891–895  
 random numbers, 914–917  
 real numbers, 891. *See also* Floating-point

- results, plausibility checking, 891  
rounding errors, 891  
rows, 895–896  
size, 891–895  
**sizeof()**, 892  
smallest integer, finding, 917  
standard mathematical functions, 917–918,  
    1181–1182  
truncation, 893  
**valarray**, 1183  
    whole numbers. *See* Integers  
Nygaard, Kristen, 833–835
- O**
- .**obj** file suffix, 48  
Object, 60, 1220  
    aliases. *See* References  
    behaving like a function. *See* Function object  
    constructing, 184  
    copying, 1115, 1119  
    current (**this**), 317  
**Date** example, 334–338  
initializing, 327–330. *See also* Constructors  
layout in memory, 308–309, 506–507  
lifetime, 1085–1086  
named. *See* Variables  
**Shape** example, 495  
**sizeof()**, 590–591  
state, 2, 305  
type, 77–78  
value. *See* Values  
Object code, 48, 1220. *See also* Executable code  
Object-oriented programming, 1220  
    “from day one,” 10  
    *vs.* generic programming, 682  
    for graphics, benefits of, 513–514  
    history of, 816, 834  
**oct** manipulator, 382–383, 1174  
Octal number system, 381–383, 1077–1078  
Off-by-one error, 149  
**ofstream**, 351–352  
Old-style casts, 1040  
One-dimensional (1D) matrices, 901–904  
On-time delivery, ideals, 810  
\ooo octal, character literal, 1080  
OOP. *See* Object-oriented programming  
Opaque types, 1060  
**open()**, 352, 1170  
Open modes, 389–390
- Open shapes, 455–456  
Opening files, 350–352. *See also* File I/O  
binary files, 390–393  
**binary** mode, 389  
C-style I/O, 1186  
failure to open, 389  
file streams, 350–352  
nonexistent files, 389  
open modes, 389–390  
testing after opening, 352  
**Open polyline** example, 455–456, 497  
Operations, 66–69, 305, 1220  
    chaining, 180–181  
    graphics classes, 490–491  
**operator**, 1038  
Operator overloading, 321  
    C++ standard operators, 322–323  
    restrictions, 322  
    user-defined operators, 322  
    uses for, 321–323  
Operator, 97–99  
    ! not, 1087  
    != not-equal (inequality), 1088  
    & (unary) address of, 588, 1087  
    & (binary) bitwise and, 956, 1089, 1094  
    && logical and, 1089, 1094  
    &= and and assign, 1090  
    % remainder (modulo), 1088  
    %&= remainder (modulo) and assign, 1090  
    \* (binary) multiply, 1088  
    \* (unary) object contents, pointing to, 1087  
    \*&= multiply and assign, 1089  
    + add (plus), 1088  
    ++ increment, 1087  
    += add and assign, 1090  
    - subtract (minus), 65, 1088  
    -- decrement, 66, 1087, 1141  
    -> (arrow) member access, 608, 1087, 1109,  
        1141  
    . (dot) member access, 1086–1087  
    / divide, 1088  
    /= divide and assign, 1090  
    :: scope resolution, 1086  
    < less than, 1088  
    << shift left, 1088. *See also* **ostream**  
    <<= shift left and assign, 1090  
    <= less than or equal, 1088  
    = assign, 1089  
    == equal, 1088  
    > greater than, 1088

Operator, *continued*

- =** greater than or equal, 1088
- >>** shift right, 1088. *See also istream*
- >>=** shift right and assign, 1090
- ?:** conditional expression (arithmetic if), 1089
- []** subscript, 1086
- ^** bitwise exclusive or, 1089, 1094
- ^=** xor and assign, 1090
- |** bitwise or, 1089, 1094
- |=** or and assign, 1090
- ||** logical or, 1089, 1094
- ~** complement, 1087
- additive operators, 1088
- const\_cast**, 1086, 1095
- delete**, 1087, 1094–1095
- delete[]**, 1087, 1094–1095
- dereference. *See* Contents of
- dynamic\_cast**, 1086, 1095
- expressions, 1086–1095
- new**, 1087, 1094–1095
- reinterpret\_cast**, 1086, 1095
- sizeof**, 1087, 1094
- static\_cast**, 1086, 1095
- throw**, 1090
- typeid**, 1086
- Optimization, laws of, 931
- or**, synonym for **|**, 1038
- Order of evaluation, 291–292
- or\_eq**, synonym for **|=**, 1038
- ostream**, 347–349, 1168–1169
  - <<**, text output, 851, 855
  - <<**, user-defined, 363–365
  - binary I/O, 390–393
  - connecting to output device, 1170
  - file I/O, **fstream**, 349–354, 1170
  - stringstreams**, 395
  - using together with stdio, 1050
- <ostream>**, 1134, 1168–1169, 1173
- ostream\_iterator** type, 790–793
- ostringstream**, 394–395
- out** mode, 389, 1170
- Out-of-class member definition, 1112
- Out-of-range conditions, 595–596
- Out\_box** example, 443, 563–564
- out\_of\_range**, 149–150, 152
- Output, 1220. *See also* Input/output; I/O streams
  - devices, 346–347
  - to file. *See* File I/O, writing files
  - floating-point values, 384–385

- format specifier **%**, 1187
- formatting. *See* Input/output, formatting
- integers, 381–383
- iterator, 752, 1142
- operations, 1173
- streams. *See* I/O streams
- to string. *See* stringstream
- testing, 1001
- Output **<<**, 47, 67, 1173
  - complex**, 920, 1183
  - string**, 851
  - text output, 851, 855
  - user-defined, 363–365
- Overflow, 891–895, 1220
- Overloading, 1104–1105, 1221
  - alternative to, 526
  - C++ and C, 1026
  - on **const**, 647–648
  - linkage, 140
  - operators. *See* Operator overloading
  - and overriding, 508–511
  - resolution, 1104–1105
- Override, 508–511, 1221

## P

- Padding, C-style I/O, 1188
- pair**, 1165–1166
  - reading sequence elements, 1152–1153
  - searching, 1158
  - sorting, 1158
- Palindromes, example, 659–660
- Paradigm, 815–818, 1221
- Parameterization, function objects, 767
- Parameterized type, 682–683
- Parameters, 1221
  - functions, 47, 115
  - list, 115
  - naming, 273
  - omitting, 273
  - templates, 679–681, 687–689
- Parametric polymorphism, 682–683
- Parsers, 190, 195
  - Expression** example, 190, 197–200, 202–203
  - functions required, 196
  - grammar rules, 194–195
  - rules *vs.* tokens, 194
- Parsing
  - expressions, 190–193
  - grammar, English, 193–194

- grammar, programming, 190–193
- tokens, 190–193
- partial\_sort()**, 1157
- partial\_sort\_copy()**, 1158
- partial\_sum()**, 770, 1184
- partition()**, 1158
- Pascal language, 829–831
- Passing arguments
  - by **const** reference, 276–278, 281–284
  - copies of, 276
  - modified arguments, 278
  - by non-**const** reference, 281–284
  - by reference, 279–284
  - temporary objects, 282
  - unmodified arguments, 277
  - by value, 276, 281–284
- Patterns. *See* Regular expressions
- Performance
  - C++ and C, 1024
  - ideals, 810
  - testing, 1012–1014
  - timing, 1015–1016
- Permutations, 1160–1161
- Petersen, Lawrence, 15
- Pictures. *See* Graphics
- Pivoting, 911–912
- Pixels, 419–420
- plus()**, 1164
- Point** example, 445–447
- pointer**, 1147
- Pointers, 594. *See also* Arrays; Iterators; Memory
  - \* contents of, 594
  - \* pointer to (in declarations), 587, 1099
  - I** subscripting, 594
  - arithmetic, 651–652
  - array. *See* Pointers and arrays
  - casting. *See* Type conversion
  - to class objects, 606–608
  - conversion. *See* Type conversion
  - to current object, **this**, 618–620
  - debugging, 656–659
  - declaration, C-style strings, 1049–1050
  - decrementing, 651–652
  - definition, 587–588, 1221
  - deleted, 657–658
  - explicit type conversion. *See* Type conversion
  - to functions, 1034–1036
  - incrementing, 651–652
  - initializing, 596–598, 657
  - vs.* iterators, 1140
- literal (**0**), 1081
- to local variables, 658
- moving around, 651
- to nonexistent elements, 657–658
- null, **0**, 598, 656–657, 1081
- NULL** macro, 1190
- vs.* objects pointed to, 593–594
- out-of-range conditions, 595–596
- palindromes, example, 661–662
- ranges, 595–596
- reading and writing through, 594–596
- semantics, 637
- size, getting, 590–591
- subscripting **I**, 594
- this**, 676–677
- unknown, 608–610
- void\***, 608–610
- Pointers and arrays
  - converting array names to, 653–654
  - pointers to array elements, 650–652
- Pointers and inheritance
  - polymorphism, 951–954
  - a problem, 944–948
  - a solution, 947–951
  - user-defined interface class, 947–951
  - vector** alternative, 947–951
- Pointers and references
  - differences, 610–611
  - inheritance, 612–613
  - list example, 613–622
  - parameters, 611–612
  - this** pointer, 618–620
- polar()**, 920, 1183
- Polar coordinates, 920, 1183
- Polygon** example, 427–428, 458–460, 497
  - vs.* **Closed\_polyline**, 458
- invariants, 460
- Polyline example
  - closed, 456–458
  - marked, 474–476
  - open, 455–456
  - vs.* rectangles, 429–431
- Polymorphism
  - ad hoc, 682–683
  - embedded systems, 951–954
  - parametric, 682–683
  - run-time, 504–505
  - templates, 682–683
- Pools, embedded systems, 940–941
- Pop-up menus, 572

- `pop_back()`**, 1149
- `pop_front()`**, 1149
- `pop_heap()`**, 1160
- Portability, 11
  - C++, 1075
  - FLTK, 418, 1204
- Positioning in files, 393–394
- Post-conditions, 165–166, 1001–1002, 1221. *See also* Invariants
- Post-decrement **`--`**, 1086, 1101
- Post-increment **`++`**, 1086, 1101
- Postfix expressions, 1086
- Pre-conditions, 163–165, 1001–1002, 1221. *See also* Invariants
- Pre-decrement **`--`**, 1087, 1101
- Pre-increment **`++`**, 1087, 1101
- Precedence, in expressions, 1090
- Precision, numeric, 386–387, 891–895
- Predicates, 763
  - on class members, 767–768
  - function objects, 1163
  - passing. *See* Function objects
  - searching, 763–764
- Predictability, 931
  - error handling, 933–934
  - features to avoid, 932
  - memory allocation, 936, 940
- Preprocessing, 265
- Preprocessor directives
  - #define**, macro substitution, 1129
  - #ifdef**, 1058–1059
  - #ifndef**, 1059
  - #include**, including headers, 1129
- Preprocessor, 1128
  - coding standards, 978–979
- `prev_permutation()`**, 1161
- Princeton University, 838
- `print`**, character class, 878, 1179
- Printable characters, identifying, 397
- `printf()` family**
  - %**, conversion specification, 1187
  - conversion specifications, 1188–1189
  - `gets()`**, 1052, 1190–1191
  - output formats, user-defined types, 1189–1190
  - padding, 1188
  - `printf()`**, 1050–1051, 1187
  - `scanf()`**, 1052–1053, 1190
  - `stderr`**, 1189
  - `stdin`**, 1189
  - `stdio`**, 1190–1191
  - `stdout`**, 1189
- synchronizing with I/O streams, 1050–1051
- truncation, 1189
- Printing
  - error messages, 150–151
  - variable values, 246
- `priority_queue`** container adaptor, 1144
- Private, 312
  - base classes, 511
  - implementation details, 210, 306–308, 312–313
  - members, 492–493, 505, 511
  - private:** label, 306, 1037
- Problem analysis, 175
  - development stages, 176
  - estimating resources, 177
  - problem statement, 176–177
  - prototyping, 178
  - strategy, 176–178
- Problem statement, 176–177
- Procedural programming languages, 815–816
- Programmers. *See also* Programming
  - communication skills, 22
  - computation ideals, 92–94
  - skills requirements, 22–23
  - stereotypes of, 21–22
  - worldwide numbers of, 843
- Programming, xxiii, 1221. *See also* Computation; Software
  - abstract-first approach, 10
  - analysis stage, 35
  - bottom-up approach, 9
  - C first approach, 9
  - concept-based approach, 6
  - concrete-first approach, 6
  - depth-first approach, 6
  - design stage, 35
  - environments, 52
  - feedback, 36
  - generic, 1219
  - implementation, 36
  - magical approach, 10
  - object-oriented, 10, 1220
  - programming stage, 36
  - software engineering principles first approach, 10
  - stages of, 35–36
  - testing stage, 36
  - top-down approach, 9–10
  - writing a program. *See* Calculator example
- Programming languages, 818–819, 821, 843
  - Ada, 832–833
  - Algol60, 827–829

- Algol family, 826–829  
assemblers, 820  
auto codes, 820  
BCPL, 838–839  
C, 836–839  
C#, 831  
C++, 839–842  
COBOL, 823–825  
Common Lisp, 825  
Delphi, 831  
Fortran, 821–823  
Lisp, 825–826  
Pascal, 829–831  
Scheme, 825  
Simula, 833–835  
Turbo Pascal, 831  
Programming philosophy, 807, 1221. *See also* C++ and C; Programming ideals; Programming languages  
Programming ideals  
abstraction level, 812–813  
aims, 807–809  
bottom-up approach, 811  
code structure, 810–811  
consistency, 814–815  
correct approaches, 811  
correctness, 810  
data abstraction, 816  
desirable properties, 807–808  
direct expression of ideas, 811–812  
efficiency, 810  
generic programming, 816  
KISS, 815  
maintainability, 810  
minimalism, 814–815  
modularity, 813–814  
multi-paradigm, 818  
object-oriented programming, 815–818  
overview, 808–809  
paradigms, 815–818  
performance, 810  
philosophies, 807–809  
procedural, 815–816  
styles, 815–818  
on-time delivery, 810  
top-down approach, 811  
Programming, history, 818–819. *See also* Programming languages  
BNF (Backus-Naur) Form, 823, 828  
classes, 834  
CODASYL committee, 824  
early languages, 819–821  
first documented bug, 824–825  
first modern stored program, 819–821  
first programming book, 820  
functional programming, 823  
function calls, 820  
inheritance, 834  
K&R, 838  
lint, 836  
object-oriented design, 834  
STL (Standard Template Library), 841  
virtual functions, 834  
Programs, 44, 1221. *See also* Computation; Software audiences for, 46  
compiling. *See* Compilers  
computing values. *See* Expression  
conforming, 1075  
experimental. *See* Prototyping  
flow, tracing, 72  
implementation defined, 1075  
legal, 1075  
linking, 51  
non-conforming constructs, 1075  
run. *See* Command line; Visual Studio, 52  
starting execution, 46–47, 1075–1076  
stored on a computer, 109  
subdividing, 177–178  
terminating, 208–209, 1075–1076  
text of. *See* Source code  
translation units, 51  
troubleshooting. *See* Debugging  
unspecified constructs, 1075  
valid, 1075  
writing, example. *See* Calculator example  
writing your first, 45–47  
Program organization. *See also* Programming ideals  
abstraction, 92–93  
divide and conquer, 93  
Projects, Visual Studio, 1199–1200  
Promotions, 99, 1091  
Prompting for input, 61  
**>**, input prompt, 223  
calculator example, 179  
sample code, 223–224  
Proofs, testing, 992  
**protected**, 492–493, 505, 511, 1037  
Prototyping, 178  
Pseudo code, 179, 1221  
Public, 306, 1037  
base class, 508  
interface, 210, 496–499

Public, *continued*  
 member, 306  
 public by default, **struct**, 307–308  
**public:** label, 306  
**punct**, punctuation character class, 878, 1179  
**Punct\_stream** example, 401–405  
 Pure virtual functions, 495, 1221  
**push\_back()**  
 growing a **vector**, 119–120  
 queue operations, 1149  
 resizing **vector**, 674–675  
 stack operations, 1149  
 string operations, 1177  
**push\_front()**, 1149  
**push\_heap()**, 1160  
**put()**, 1173  
**putback()**  
 naming convention, 211  
 putting tokens back, 206–207  
 return value, disabling, 211–212  
**putc()**, 1191  
**putchar()**, 1191  
 Putting back input, 206–208

## Q

**qsort()**, 1194–1195  
**<queue>**, 1134  
**queue** container adaptor, 1144  
 Queue operations, 1149

## R

**\r** carriage return, character literal, 1079  
**r**, reading file mode, 1186  
**r+**, reading and writing file mode, 1186  
 RAII (Resource Acquisition Is Initialization)  
 definition, 1221  
 exceptions, 700–701, 1125  
 testing, 1004–1005  
 for **vector**, 705–707  
**<random>**, 1134  
 Random numbers, 914–917  
 Random-access iterators, 752, 1142  
 Range  
 definition, 1221  
 errors, 148–150  
 pointers, 595–596  
 regular expressions, 877–878

Range checking  
**at()**, 693–694  
**I**, 650–652, 693–696  
 arrays, 650–652  
 compatibility, 695  
 constraints, 695  
 design considerations, 694–696  
 efficiency, 695  
 exceptions, 693–694  
 macros, 696–697  
 optional checking, 695–696  
 overview, 693–694  
 pointer, 650–652  
**vector**, 693–696  
range-for, 119  
**rbegin()**, 1148  
 Re-throwing exceptions, 702, 1126  
**read()**, unformatted input, 1172  
 Readability  
 expressions, 95  
 indenting nested code, 271  
 nested code, 271  
 Reading  
 dividing functions logically, 359–362  
 files. *See* Reading files  
 with iterators, 1140–1141  
 numbers, 214–215  
 potential problems, 358–363  
 separating dialog from function, 362–363  
 a series of values, 356–358  
 a single value, 358–363  
 into **strings**, 851  
 tokens, 185  
 Reading files  
 binary I/O, 391  
 converting representations, 374–376  
 to end of file, 366  
 example, 352–354  
**fstream** type, 350–352  
**ifstream** type, 350–352  
 input loops, 365–367  
**istream** type, 349–354, 391  
 in-memory representation, 368–370  
**ostream** type, 391  
 process steps, 350  
 structured files, 367–376  
 structured values, 370–374  
 symbolic representations, 374–376  
 terminator character, specifying, 366

- real()**, 920, 1183  
Real numbers, 891  
Real part, 920  
Real-time constraints, 931  
Real-time response, 928  
**realloc()**, 1045, 1193  
Recovering from errors, 239–241, 355–358. *See also* Error handling; Exceptions  
**Rectangle** example, 428–431, 460–465, 497  
Recursion  
    definition, 1221  
    infinite, 198, 1220  
    looping, 200  
Recursive function calls, 289  
Red-black trees, 779. *See also* Associative containers; **map**, associative array  
Red margin alerts, 3  
Reference semantics, 637  
References, 1221. *See also* Aliases  
    & in declarations, 276–279  
    to arguments, 277–278  
    circular. *See* Circular reference  
    to last **vector** element, **back()**, 737  
    vs. pointers. *See* Pointers and references  
**<regex>**, 1134, 1175  
**regex**. *See* Regular expressions  
**regex\_error** exception, 1138  
**regex\_match()**, 1177  
    vs. **regex\_search()**, 883  
**regex\_search()**, 1177  
    vs. **regex\_match()**, 883  
**regex** pattern matching, 866–868  
    \$ end of line, 873, 1178  
    () grouping, 867, 873, 876  
    \* zero or more occurrences, 868, 873–874  
    [] character class, 873  
    \ escape character, 866–867, 873  
    \ as literal, 877  
    ^ negation, 873  
    ^ start of line, 873  
    {} count, 867, 873–875  
    | alternative (or), 867–868, 873, 876  
    + one or more occurrences, 873, 874–875  
    . wildcard, 873  
    ? optional occurrence, 867–868, 873, 874–875  
    alternation, 876  
    character classes. *See* **regex** character classes  
        character sets, 877–878  
        definition, 870  
        grouping, 876  
        **matches**, 870  
        pattern matching, 872–873  
        ranges, 877–878  
        **regex** operators, 873, 1177–1179  
        **regex\_match()**, 1177  
        **regex\_search()**, 1177  
        repeating patterns, 874–876  
        searching with, 869–872, 880  
        **smatch**, 870  
        sub-patterns, 867, 870  
    **regex** character classes, 877–878  
    **alnum**, 878  
    **alpha**, 878  
    **blank**, 878  
    **cntrl**, 878  
    **d**, 878  
    **\d**, 873  
    **\D**, 873  
    **digit**, 878  
    **graph**, 878  
    **\L**, 873  
    **\L**, 874  
    **lower**, 878  
    **print**, 878  
    **punct**, 878  
    **regex\_match()** vs. **regex\_search()**, 883  
    **s**, 878  
    **\s**, 873  
    **\S**, 874  
    **space**, 878  
    **\u**, 873  
    **\U**, 874  
    **upper**, 878  
    **w**, 878  
    **\w**, 873  
    **\W**, 873  
    **xdigit**, 878  
Regression tests, 993  
Regular expressions, 866–868, 872, 1221.  
    *See also* **regex** pattern matching  
    character classes, 873–874  
    error handling, 878–880  
    grouping, 867, 873, 876  
    uses for, 865  
    ZIP code example, 880–885  
Regularity, 380

**reinterpret\_cast**, 609–610, 1095  
 casting unrelated types, 609  
 hardware access, 944  
**Relational operators**, 1088  
**Reliability**, software, 34, 928  
**Remainder and assign %=**, 1090  
**Remainder %** (modulo), 66, 1088  
 correspondence to \* and /, 68  
 floating-point, 201, 230–231  
 integer and floating-point, 66  
**remove()**, 1155  
**remove\_copy()**, 1155  
**remove\_copy\_if()**, 1155  
**rend()**, 1148  
 Repeated words examples, 71–74  
 Repeating patterns, 194  
 Repetition, 1178. *See also* Iteration; **regex**  
**replace()**, 1155  
**replace\_copy()**, 1155  
 Reporting errors  
**Date** example, 317–318  
 debugging, 159  
**error()**, 142–143  
 run-time, 145–146  
 syntax errors, 137–138  
 Representation, 305, 671–673  
 Requirements, 1221. *See also* Invariants; Post-conditions; Pre-conditions  
 for functions, 153  
**reserve()**, 673–674, 691, 747, 1151  
 Reserved names, 75–76. *See also* Keywords  
**resetiosflags()** manipulator, 1174  
**resize()**, 674, 1151  
 Resource, 1221  
 leaks, 931, 934  
 limitations, 928  
 management. *See* Resource management  
 testing, 1001–1002  
**vector** example, 697–698  
 Resource Acquisition Is Initialization (RAII), 1221  
 exceptions, 700–701, 1125  
 testing, 1004–1005  
 for **vector**, 705–707  
 Resource management, 697–702. *See also* **vector**  
 example  
 basic guarantee, 702  
 error handling, 702  
 guarantees, 701–702  
**make\_vec()**, 702  
 no-throw guarantee, 702  
 problems, 698–700  
 RAII, 700–701, 705–707  
 resources, examples, 697–698  
 strong guarantee, 702  
 testing, 1004–1005  
 Results, 91. *See also* Return values  
**return** and move, 704–705  
**return** statement, 272–273  
 Return types, functions, 47, 272–273  
 Return values, 113–115  
 functions, 1103  
 no return value, **void**, 212  
 omitting, 115  
 returning, 272–273  
**reverse()**, 1155  
**reverse\_copy()**, 1155  
**reverse\_iterator**, 1147  
 Revision history, 237–238  
 Rho, 920  
 Richards, Martin, 838  
**right** manipulator, 1174  
 Ritchie, Dennis, 836, 837, 842, 1022–1023, 1032  
 Robot-assisted surgery, 30  
**rotate()**, 1155  
**rotate\_copy()**, 1155  
 Rounding, 386, 1221. *See also* Truncation  
 errors, 891  
 floating-point values, 386  
 Rows, matrices, 900–901, 906  
 Rules, for programming. *See* Ideals  
 Rules, grammatical, 194–195  
 Run-time dispatch, 504–505. *See also* Virtual functions  
 Run-time errors. *See* Errors, run-time  
 Run-time polymorphism, 504–505  
**runtime\_error**, 142, 151, 153  
 rvalue reference, 639  
 Rvalues, 94–95, 1090

## S

**s**, character class, 878, 1179  
**\S**, “not space,” **regex**, 874  
**\s**, “space,” **regex**, 873  
 Safe conversions, 79–80  
 Safety, type. *See* Type, safety  
 Scaffolding, cleaning up, 234–235  
**scale\_and\_add()** example, 904  
**scale\_and\_multiply()** example, 912  
 Scaling data, 542–543

- scanf()**, 1052, 1190  
Scenarios. *See* Use cases  
Scheme language, 825  
**scientific** format, 387  
**scientific** manipulator, 385, 1174  
Scope, 266–267, 1082–1083, 1221  
    class, 267, 1082  
    enumerators, 320–321  
    global, 267, 270, 1082  
    going out of, 268–269  
    kinds of, 267  
    local, 267, 1083  
    namespace, 267, 271, 1082  
    resolution `::`, 295–296, 1086  
    statement, 267, 1083  
Scope and nesting  
    blocks within functions, 271  
    classes within classes, 270  
    classes within functions, 270  
    functions within classes, 270  
    functions within functions, 271  
    indenting nested code, 271  
    local classes, 270  
    local functions, 270  
    member classes, 270  
    member functions, 270  
    nested blocks, 271  
    nested classes, 270  
    nested functions, 270  
Scope and object lifetime, 1085–1086  
    free-store objects, 1085  
    local (automatic) objects, 1085  
    namespace objects, 1085  
    static class members, 1085  
    temporary objects, 1085  
Scope and storage class, 1083–1084  
    automatic storage, 1083–1084  
    free store (heap), 1084  
    static storage, 1084  
Screens. *See also* GUIs (graphical user interfaces)  
    data graph layout, 541–542  
    drawing on, 423–424  
    labeling, 425  
**search()**, 795–796, 1153  
Searching. *See also* Finding; Matching; **find\_if()**; **find()**  
    algorithms for, 1157–1159  
    binary searches, 779, 795–796  
    in C, 1194–1195  
    for characters, 740  
(key,value) pairs, by key. *See* Associative containers  
for links, 615–617  
**map** elements. *See* **unordered\_map**  
predicates, 763  
with regular expressions, 869–872, 880–885, 1177–1179  
**search\_n()**, 1153  
Self reference. *See* **this** pointer  
Self assignment, 676–677  
Self-checking, error handling, 934  
Separators, nonstandard, 398–405  
Sequence containers, 1144  
Sequences, 720, 1221  
    algorithms. *See* Algorithms, STL  
    differences between adjacent elements, 770  
    empty, 729  
    example, 723–724  
    half open, 721  
    Sequencing rules, 195  
Server farms, 31–32  
**set**, 776, 787–789  
    iterators, 1144  
    *vs.* **map**, 788  
    subscripting, 788  
**set()**, 605–606  
**<set>**, 776, 1134  
Set algorithms, 1159–1160  
**set\_difference()**, 1160  
**set\_intersection()**, 1159  
**set\_symmetric\_difference()**, 1160  
**set\_union()**, 1159  
**setbase()** manipulator, 1174  
**setfill()** manipulator, 1174  
**setiosflags()** manipulator, 1174  
**setprecision()** manipulator, 386–387, 1174  
**setw()** manipulator, 1174  
Shallow copies, 636  
**Shape** example, 493–494  
    abstract classes, 495–496  
    access control, 496–499  
    attaching to **Window**, 545–546  
    as base class, 445, 495–496  
    **clone()**, 504  
    copying objects, 503–504  
    **draw()**, 500–502  
    **draw\_lines()**, 500–502  
    fill color, 500  
    implementation inheritance, 513–514  
    interface inheritance, 513–514

- Shape** example, *continued*  
 line visibility, 500  
**move()**, 502  
 mutability, 503–504  
**number\_of\_points()**, 449  
 object layout, 506–507  
 object-oriented programming, 513–514  
**point()**, 449  
 slicing shapes, 504  
 virtual function calls, 501, 506–507  
 Shift operators, 1088  
 Shipping, computer use, 26–28  
**short**, 955, 1099  
 Shorthand notation, regular expressions, 1179  
**showbase**, manipulator, 383, 1173  
**showpoint**, manipulator, 1173  
**showpos**, manipulator, 1173  
 Shuffle algorithm, 1155–1156  
 Signed and unsigned integers, 961–965  
**signed** type, 1099  
**Simple\_window**, 422–424, 443  
 Simplicity ideal, 92–94  
 Simula language, 833–835  
**sin()**, sine, 917, 1182  
 Singly-linked lists, 613, 725  
**sinh()**, hyperbolic sine, 918, 1182  
 Size  
   bit strings, 955–956  
   containers, 1150–1151  
   getting, **sizeof()**, 590–591  
   of numbers, 891–895  
   **vectors**, getting, 119–120  
**size()**  
   container capacity, 1150  
   number of elements, 120, 851  
   string length, 851, 1176  
   **vectors**, 120, 122–123  
**sizeof()**, 590–591, 1094  
   object size, 1087  
   value size, 892  
**size\_type**, 730, 1147  
**skipws**, 1174  
**slice()**, 901–902, 905  
 Slicing  
   matrices, 901–902, 905  
   objects, 504  
 Smallest integer, finding, 917  
**smatch**, 870  
 Soft real-time, 931  
 Software, 19, 1222. *See also* Programming; Programs  
   affordability, 34  
   correctness, 34  
   ideals, 34–37  
   maintainability, 35  
   reliability, 34  
   troubleshooting. *See* Debugging  
   useful design, 34  
   uses for, 19–33  
 Software layers, GUIs, 557  
**sort()**, 758, 794–796, 1157  
**sort\_heap()**, 1160  
 Sorting  
   algorithms for, 1157–1159  
   in C, **qsort()**, 1194  
   **sort()**, 758, 794–796, 1157  
 Source code  
   definition, 48, 1222  
   entering, 1200  
 Source files, 48, 1222  
   adding to projects, 1200  
**space**, 878, 1179  
 Space exploration, computer use, 33  
 Special characters, 1079–1080  
   regular expressions, 1178  
 Specialization, 681, 1123  
 Specifications  
   definition, 1221  
   source of errors, 136  
 Speed of light, 96  
**sprintf()**, 1187  
**sqrt()**, square root, 917, 1181  
 Square of **abs()**, norm, 919  
**<sstream>**, 1134  
**stable\_partition()**, 1158  
**stable\_sort()**, 1157  
**<stack>**, 1134  
**stack** container adaptor, 1144  
 Stack of activation records, 287  
 Stack storage, 591–592  
 Stacks  
   container operations, 1149  
   embedded systems, 935–936, 940, 942–943  
   growth, 287–290  
   unwinding, 1126  
 Stages of programming, 35–36  
 Standard  
   conformance, 836, 974, 1075  
   ISO, 1075, 1222

manipulators. *See* Manipulators  
mathematical functions, 917–918  
Standard library. *See also* C standard library; STL  
(Standard Template Library)  
algorithms. *See* Algorithms  
**complex**. *See* complex  
containers. *See* Containers  
C-style I/O. *See* printf() family  
C-style strings. *See* C-style strings  
date and time, 1193–1194  
function objects. *See* Function objects  
I/O streams. *See* Input; Input/output;  
    Output  
iterators. *See* Iterators  
mathematical functions. *See* Mathematical  
    functions (standard)  
numerical algorithms. *See* Algorithms,  
    numerical; Numerics  
**string**. *See* string  
time, 1015–1016, 1193  
**valarray**. *See* valarray  
Standard library header files, 1133–1136  
algorithms, 1133–1134  
containers, 1133–1134  
C standard libraries, 1135–1136  
I/O streams, 1134  
iterators, 1133–1134  
numerics, 1134–1135  
string manipulation, 1134  
utility and language support, 1135  
Standard library I/O streams, 1168–1169. *See also*  
    I/O streams  
Standard library string manipulation  
    character classification, 1175–1176  
    containers. *See* map, associative array; set;  
        unordered\_map; vector  
    input/output. *See* I/O streams  
    regular expressions. *See* regex  
    string manipulation. *See* string  
Stanford University, 826  
Starting programs, 1075–1076. *See also* main()  
State, 90–91, 1222  
    I/O stream, 1171  
    of objects, 305  
    source of errors, 136  
    testing, 1001  
    validity checking, 313  
    valid state, 313  
Statement scope, 267, 1083  
Statements, 47  
    grammar, 1096–1097  
    named sequence of. *See* Function  
    terminator ; (semicolon), 50, 100  
Static storage, 591–592, 1084  
    class members, lifetime, 1085  
    embedded systems, 935–936, 944  
**static**, 1084  
**static const**, 326. *See also* const  
**static** local variables, order of initialization, 294  
**std** namespace, 296–297, 1136  
**stderr**, 1189  
**<stdexcept>**, 1135  
**stdin**, 1050, 1189. *See also* stdio  
stdio, standard C I/O, 1050, 1190–1191  
**EOF** macro, 1053–1054  
**errno**, error indicator, 918–919  
**fclose()**, 1053–1054  
**FILE**, 1053–1054  
**fopen()**, 1053–1054  
**getchar()**, 1052–1053, 1191  
**gets()**, 1052, 1190–1191  
input, 1052–1053  
output, 1050–1051  
**printf()**, 1050–1051, 1188–1191  
**scanf()**, 1052, 1190  
**stderr**, **cerr** equivalent, 1189  
**stdin**, **cin** equivalent, 1050, 1189  
**stdout**, 1050, 1189. *See also* stdio  
**stdout**, **cout** equivalent, 1050, 1189  
**std\_lib\_facilities.h** header file, 1199–1200  
**stdout**, 1050, 1189. *See also* stdio  
Stepanov, Alexander, 720, 722, 841  
Stepping through code, 162  
Stereotypes of programmers, 21–22  
STL (Standard Template Library), 717, 1149–  
    1168 (large range, not sure this is correct). *See*  
    also C standard library; Standard library  
algorithms. *See* STL algorithms  
containers. *See* STL containers  
function objects. *See* STL function objects  
history of, 841  
ideals, 717–720  
iterators. *See* STL iterators  
namespace, **std**, 1136  
STL algorithms, 1152–1162  
    *See* Algorithms, STL  
alternatives to, 1195  
built-in arrays, 747–749

STL algorithms, *continued*  
 computation *vs.* data, 717–720  
 heap, 1160  
**max()**, 1161  
**min()**, 1161  
 modifying sequence, 1154–1156  
 mutating sequence, 1154–1156  
 nonmodifying sequence, 1153–1154  
 permutations, 1160–1161  
 searching, 1157–1159  
 set, 1159–1160  
 shuffle, 1155–1156  
 sorting, 1157–1159  
 utility, 1157  
 value comparisons, 1161–1162

STL containers, 749–751, 1144–1152  
 almost, 751, 1145  
 assignments, 1148  
 associative, 1144, 1151–1152  
 capacity, 1150–1151  
 comparing, 1151  
 constructors, 1148  
 container adaptors, 1144  
 copying, 1151  
 destructors, 1148  
 element access, 1149  
 information sources about, 750  
 iterator categories for, 752, 1143–1145,  
     1148  
 list operations, 1150  
 member types, 1147  
 operations overview, 1146–1147  
 queue operations, 1149  
 sequence, 1144  
 size, 1150–1151  
 stack operations, 1149  
 swapping, 1151

STL function objects, 1163  
 adaptors, 1164  
 arithmetic operations, 1164  
 inserters, 1162–1163  
 predicates, 767–768, 1163

STL iterators, 1139–1140  
 basic operations, 721  
 categories, 1142–1143  
 definition, 721, 1139  
 description, 721–722  
 empty lists, 729  
 example, 737–741

operations, 1141–1142  
*vs.* pointers, 1140  
 sequence of elements, 1140–1141

Storage class, 1083–1084  
 automatic storage, 1083–1084  
 free store (heap), 1084  
 static storage, 1084

Storing data. *See* Containers

**str()**, **string** extractor, 395  
**strcat()**, 1047, 1191  
**strchr()**, 1048, 1192  
**strcmp()**, 1047, 1192  
**strcpy()**, 1047, 1049, 1192

Stream  
 buffers, 1169  
 iterators, 790–793  
 modes, 1170  
 states, 355  
 types, 1170  
**streambuf**, 406, 1169  
**<streambuf>**, 1134  
**<string>**, 1134, 1172  
**string**, 66, 851, 1222. *See also* Text  
 [] subscripting, 851  
 + concatenation, 68–69, 851, 1176  
 += append, 851  
 < lexicographical comparison, 851  
 == equal, 851  
 = assign, 851  
 >> input, 851  
 << output, 851  
 almost container, 1145  
**append()**, 851  
**basic\_string**, 852  
 C++ to C-style conversion, 851  
**c\_str()**, C++ to C-style conversion, 851  
**erase()**, removing characters, 851  
 exceptions, 1138  
**find()**, 851  
**from\_string()**, 853–854  
**getline()**, 851  
 input terminator (whitespace), 65  
**Insert()**, adding characters, 851  
**length()**, number of characters, 851  
**lexical\_cast** example, 855  
 literals, debugging, 161  
 operations, 851, 1176–1177  
 operators, 66–67, 68  
 palindromes, example, 659–660

pattern matching. *See* Regular expressions  
 properties, 741–742  
 size, 78  
`size()`, number of characters, 851  
 standard library, 852  
`stringstream`, 852–854  
`string` to value conversion, 853–854  
 subscripting  $\text{[ ]}$ , 851  
`to_string()` example, 852–854  
 values to string conversion, 852  
*vs.* `vector`, 745  
 whitespace, 854  
 String literal, 62, 1080  
`stringstream`, 395, 852–854, 1170  
`strlen()`, 1046, 1191  
`strncat()`, 1047, 1192  
`strcmp()`, 1047, 1192  
`strcpy()`, 1047, 1192  
 Strong guarantee, 702  
 Stroustrup, Bjarne  
   adviser, 820  
   Bell Labs colleagues, 836–839, 1023  
   biography, 13–14  
   education on invariants, 828  
   inventor of C++, 839–842  
   Kristen Nygaard, 834  
`strpbrk()`, 1192  
`strrchr()`, 1192  
`strstr()`, 1192  
`strtod()`, 1192  
`strtol()`, 1192  
`strtoul()`, 1192  
`struct`, 307–308. *See also* Data  
`struct` tag namespace, 1036–1037  
 Structure  
   of data. *See* Data  
   of programs, 215–216  
 Structured files, 367–376  
 Style, definition, 1222  
 Sub-patterns, 867, 870  
 Subclasses, 504. *See also* Derived classes  
 Subdividing programs, 177–178  
 Subscripting, 118  
    $\text{[ ]}$  Fortran style, 899  
    $\text{[ ]}$  C Style, 694, 899  
   arrays, 649, 899  
   `at()`, checked subscripting, 694, 1149  
   `Matrix` example, 899–901, 905  
   pointers, 1101  
`string`, 851, 1176  
`vector`, 594, 607–608, 646–647  
 Substrings, 863  
 Subtraction  $-$  (minus)  
`complex`, 919, 1183  
 definition, 1088  
 integers, 1101  
 iterators, 1141–1142  
 pointers, 1101  
 Subtype, definition, 1222  
 Summing values. *See* `accumulate()`  
 Superclasses, 504, 1222. *See also* Base classes  
`swap()`, 281, 1151, 1157  
 Swapping  
   columns, 906  
   containers, 1151  
   ranges, 1157  
   rows, 906, 912  
`swap_ranges()`, 1157  
`switch`-statements  
   `break`, `case` termination, 106–108  
   `case` labels, 106–108  
   most common error, 108  
   *vs.* `string`-based selection, 106  
 Symbol tables, 247  
 Symbolic constants. *See also* Enumerations  
   cleaning up, 232–234  
   defining, with `static const`, 326  
 Symbolic names, tokens, 233  
 Symbolic representations, reading, 374–376  
 Syntax analyzers, 190  
 Syntax checking, 48–50  
 Syntax errors  
   examples, 48–50  
   overview, 137–138  
   reporting, 137–138  
 Syntax macros, 1058  
`system()`, 1194  
`system_clock`, 1016, 1185  
 System, definition, 1222  
 System tests, 1009–1011

**T**

`\t` tab character, 109, 1079  
`tan()`, tangent, 917, 1182  
`tanh()`, hyperbolic tangent, 917, 1182  
 TEA (Tiny Encryption Algorithm), 820, 969–974  
 Technical University of Copenhagen, 828

- Telecommunications, 28–29
- Temperature data, example, 120–123
- template**, 1038
- Template, 678–679, 1121–1122, 1222
  - arguments, 1122–1123
  - class, 681–683. *See also* Class template
  - compiling, 684
  - containers, 686–687
  - error diagnostics, 683
  - function, 682–690. *See also* Function template
  - generic programming, 682–683
  - inheritance, 686–687
  - instantiation, 681, 1123–1124
  - integer parameters, 687–689
  - member types, 1124
  - parameters, 679–681, 687–689
  - parametric polymorphism, 682–683
  - specialization, 1123
  - typename**, 1124
  - type parameters, 679–681
  - weaknesses, 683
- Template-style casts, 1040
- Temporary objects, 282, 1085
- Terminals, in grammars. *See* Tokens
- Termination
  - abort()** a program, 1194
  - on exceptions, 142
  - exit()** a program, 1194
  - input, 61–62, 179
  - normal program termination, 1075–1076
  - for **string** input, 65
  - zero, for C-style strings, 654–655
- Terminator character, specifying, 366
- Testing, 992–993, 1222. *See also* Debugging
  - algorithms, 1001–1008
  - for bad input, 103
  - black box, 992–993
  - branching, 1006–1008
  - bug reports, retention period, 993
  - calculator example, 225
  - code coverage, 1008
  - debugging, 1012
  - dependencies, 1002–1003
  - designing for, 1011–1012
  - faulty assumptions, 1009–1011
  - files, after opening, 352
  - FLTK, 1206
  - inputs, 1001
  - loops, 1005–1006
  - non-algorithms, 1001–1008
  - outputs, 1001
  - performance, 1012–1014
  - pre- and post-conditions, 1001–1002
  - proofs, 992
  - RAII, 1004–1005
  - regression tests, 993
  - resource management, 1004–1005
  - resources, 1001–1002
  - stage of programming, 36
  - state, 1001
  - system tests, 1009–1011
  - test cases, definition, 166
  - test harness, 997–999
  - timing, 1015–1016
  - white box, 992–993
- Testing units
  - formal specification, 994–995
  - random sequences, 999–1001
  - strategy for, 995–997
  - systematic testing, 994–995
  - test harness, 997–999
- Text
  - character strings. *See* C-style strings; **string**
  - email example, 856–861, 864–865
  - extracting text from files, 855–861, 864–865
  - finding patterns, 864–865, 869–872
  - in graphics. *See* Text
  - implementation details, 861–864
  - input/output, GUIs, 563–564
  - maps. *See* map
  - storage, 591–592
  - substrings, 863
  - vector** example, 123–125
  - words frequency example, 777–779
- Text** example, 431–433, 467–470
- Text editor example, 737–741
- Theta, 920
- this** pointer, 618–620, 676–677
- Thompson, Ken, 836–838
- Three-way comparison, 1046
- Throwing exceptions, 147, 1125
  - I/O stream, 1171
  - re-throwing, 702
  - standard library, 1138–1139
  - throw, 147, 1090, 1125–1126
  - vector**, 697–698
- Time
  - date and time, 1193–1194
  - measuring, 1015–1016
- Timekeeping, computer use, 26

- time\_point**, 1016  
**time\_t**, 1193  
Tiny Encryption Algorithm (TEA), 820, 969–974  
**tm**, 1193  
**Token** example, 183–184  
**Token\_stream** example, 206–214  
**tolower()**, 398, 1176  
Top-down approach, 9–10, 811  
**to\_string()** example, 852–854  
**toupper()**, 398, 1176  
Tracing code execution, 162–163  
Trade-off, definition, 1222  
**transform()**, 1154  
Transient errors, handling, 934  
Translation units, 51, 139–140  
Transparency, 451, 463  
Tree structure, **map** container, 779–782  
**true**, 1037, 1038  
**trunc** mode, 389, 1170  
Truncation, 82, 1222  
    C-style I/O, 1189  
    exceptions, 153  
    floating-point numbers, 893  
**try-catch**, 146–153, 693–694, 1037  
Turbo Pascal language, 831  
Two-dimensional matrices, 904–906  
Two’s complement, 961  
Type, 60, 77, 1222  
    aliases, 730  
    built-in. *See* Built-in types  
    checking C++ and C, 1032–1033  
    generators, 681  
    graphics classes, 488–490  
    mismatch errors, 138–139  
    mixing in expressions, 99  
    naming. *See* Namespaces  
    objects, 77–78  
    operations, 305  
    organizing. *See* Namespaces  
    parameterized, 682–683. *See also* Template  
    as parameters. *See* Template  
    pointers. *See* Pointer  
    promotion, 99  
    representation of object, 308–309, 506–507  
    safety, 78–79, 82  
    subtype, 1222  
    supertype, 1222  
    truncation, 82  
    user-defined. *See* UDTs (user-defined types)  
    uses for, 304  
values, 77  
variables. *See* Variables  
Type conversion  
    casting, 609–610  
**const\_cast**, casting away **const**, 609–610  
exceptions, 153  
**explicit**, 609  
    in expressions, 99–100  
    function arguments, 284–285  
    implicit, 642–643  
**int** to pointer, 590  
operators, 1095  
pointers, 590, 609–610  
**reinterpret\_cast**, 609  
safety, 79–83  
**static\_cast**, 609  
**string** to value, 853–854  
truncation, 82  
    value to **string**, 852  
Type conversion, implicit, 642–643  
**bool**, 1092  
compiler warnings, 1091  
floating-point and integral, 1091–1092  
integral promotion, 1091  
pointer and reference, 1092  
preserving values, 1091  
promotions, 1091  
user-defined, 1091  
usual arithmetic, 1092  
Type safety, 78–79  
    implicit conversions, 80–83  
    narrowing conversions, 80–83  
    pointers, 596–598, 656–659  
    range error, 148–150, 595–596  
    safe conversions, 79–80  
    unsafe conversions, 80–83  
**typedef**, 730  
**typeid**, 1037, 1087, 1138  
**<typeinfo>**, 1135  
**typename**, 1037, 1124

## U

- u/U** suffix, 1077  
**\U**, “not uppercase,” **regex**, 874  
**\u**, “uppercase character,” **regex**, 873, 1179  
UDTs (user-defined types). *See* Class;  
    Enumerations  
Unary expressions, 1087  
“Uncaught exception” error, 153

Unchecked conversions, 943–944  
 “Undeclared identifier” error, 258  
 Undefined order of evaluation, 263  
**unget()**, 355–358  
**ungetc()**, 1191  
 Uninitialized variables, 327–330, 1222  
**uninitialized\_copy()**, 1157  
**uninitialized\_fill()**, 1157  
**union**, 1121  
**unique()**, 1155  
**unique\_copy()**, 758, 789, 792–793, 1155  
**unique\_ptr**, 703–704  
 Unit tests  
   formal specification, 994–995  
   random sequences, 999–1001  
   strategy for, 995–997  
   systematic testing, 994–995  
   test harness, 997–999  
 Universal and uniform initialization, 83  
 Unnamed objects, 465–467  
**<unordered\_map>**, 776, 1134  
**unordered\_map**, 776. *See also map*, associative array  
   finding elements, 785–787  
   hashing, 785  
   hash tables, 785  
   hash values, 785  
   iterators, 1144  
**unordered\_multimap**, 776, 1144  
**unordered\_multiset**, 776, 1144  
**<unordered\_set>**, 776, 1134  
**unordered\_set**, 776, 1144  
 Unsafe conversions, 80–83  
**unsetf()**, 384  
 Unsigned and signed, 961–965  
**unsigned** type, 1099  
 Unspecified constructs, 1075  
**upper**, character class, 878, 1179  
**upper\_bound()**, 796, 1152, 1158  
 Uppercase. *See Case (of characters)*  
**uppercase**, 1174  
 U.S. Department of Defense, 832  
 U.S. Navy, 824  
 Use cases, 179, 1222  
 User-defined conversions, 1091  
 User-defined operators, 1091  
 User-defined types (UDTs), 304. *See also Class*;  
   Enumerations  
   exceptions, 1126  
   operator overloading, 1107

operators, 1107  
 standard library types, 304  
 User interfaces  
   console input/output, 552  
   graphical. *See GUIs (graphical user interfaces)*  
   web browser, 552–553  
**using** declarations, 296–297  
**using** directives, 296–297, 1127  
 Usual arithmetic conversions, 1092  
 Utilities, STL  
   function objects, 1163–1164  
   inserters, 1162–1163  
**make\_pair()**, 1165–1166  
**pair**, 1165–1166  
**<utility>**, 1134, 1165–1166  
 Utility algorithms, 1157  
 Utility and language support, header files, 1135

## V

**v** vertical tab, character literal, 1079  
**valarray**, 1145, 1183  
**<valarray>**, 1135  
 Valid pointer, 598  
 Valid programs, 1075  
 Valid state, 313  
 Validity checking, 313  
   constructors, 313  
   enumerations, 320  
   invariants, 313  
   rules for, 313  
 Value semantics, 637  
**value\_comp()**, 1152  
 Values, 77–78, 1222  
   symbolic constants for. *See Enumerations*  
   and variables, 62, 73–74, 243  
**value\_type**, 1147  
 Variables, 62–63, 1083  
    $\text{++}$  increment, 73–74  
    $=$  assignment, 69–73  
   changing values, 73–74  
   composite assignment operators, 73–74  
   constructing, 291–292  
   declarations, 260, 262–263  
   going out of scope, 291  
   incrementing  $\text{++}$ , 73–74  
   initialization, 69–73  
   input, 60  
   naming, 74–77

- type of, 66–67  
 uninitialized, class interfaces, 327–330  
 value of, 73–74  
**<vector>**, 1134  
**vector** example, 584–587, 629–636, 668–679  
**[** subscripting, 646, 693–697  
**=** assignment, 675–677  
**.** (dot) access, 607–608  
 allocators, 691  
 changing size, 668–679  
**at()**, checked subscripting, 694  
 copying, 631–636  
 destructor, 601–605  
 element type as parameter, 679–681  
**erase()** (removing elements), 745–747  
 exceptions, 693–694, 705–707  
**explicit** constructors, 642–643  
 inheritance, 686–687  
**insert()** (adding elements), 745–747  
 overloading on **const**, 647–648  
**push\_back()**, 674–675, 692  
 representation, 671–673  
**reserve()**, 673, 691, 704–705  
**resize()**, 674, 692  
 subscripting, 594, 607–608, 646–647  
**vector**, standard library, 1146–1151  
**[** subscripting, 1149  
**=** assignment, 1148  
**==** equality, 1151  
**<** less than, 1151  
**assign()**, 1148  
**back()**, reference to last element, 1149  
**begin()**, iterator to first element, 1148  
**capacity()**, 1151  
**at()**, checked subscripting, 1149  
**const\_iterator**, 1147  
 constructors, 1148  
 destructor, 1148  
**difference\_type**, 1147  
**end()**, one beyond last element, 1148  
**erase()**, removing elements, 1150  
**front()**, reference to first element, 1149  
**insert()**, adding elements, 1150  
**iterator**, 1147  
 member functions, lists of, 1147–1151  
 member types, list of, 1147  
**push\_back()**, add element at end, 1149  
**size()**, number of elements, 1151  
**size\_type**, 1147  
**value\_type**, 1147  
**vector** of references, simulating, 1212–1213  
**Vector\_ref** example, 444, 1212–1213  
**vector\_size()**, 119  
**virtual**, 1037  
 Virtual destructors, 604–605. *See also* Destructors  
 Virtual functions, 501, 506–507  
 declaring, 508  
 definition, 501, 1222  
 history of, 834  
 object layout, 506–507  
 overriding, 508–511  
 pure, 512–513  
**Shape** example, 501, 506–507  
**vptr**, 506–507  
**vtbl**, 506  
 Visibility. *See also* Scope; Transparency  
 menus, 573–574  
 of names, 266–272, 294–297  
 widgets, 562  
 Visual Studio  
 FLTK (Fast Light Toolkit), 1205–1206  
 installing, 1198  
 running programs, 1199–1200  
**void**, 115  
 function results, 115, 273, 275  
 pointer to, 608–610  
**putback()**, 212  
**void\***, 608–610, 1041–1042, 1099  
**vptr**, virtual function pointer, 506–507  
**vtbl**, virtual function table, 506

## W

- w**, writing file mode, 878, 1179, 1186  
**w+**, writing and reading file mode, 1186  
**\W**, “not word character,” **regex**, 874, 1179  
**\w**, “word character,” **regex**, 873, 1179  
**wait()**, 559–560, 569–570  
 Wait loops, 559–560  
**wait\_for\_button()** example, 559–560  
 Waiting for user action, 559–560, 569–570  
**wchar\_t**, 1038  
 Web browser, as user interface, 552–553  
 Wheeler, David, 109, 820, 954, 969  
**while**-statements, 109–111  
 vs. **for**, 122  
 White-box testing, 992–993  
 Whitespace  
 formatting, 397, 398–405  
 identifying, 397

Whitespace  
in input, 64  
**string**, 854  
**Widget** example, 561–563  
**Button**, 422–424, 553–561  
control inversion, 569–570  
debugging, 576–577  
**hide()**, 562  
implementation, 1209–1210  
**In\_box()**, 563–564  
line drawing example, 565–569  
**Menu**, 564–565, 570–575  
**move()**, 562  
**Out\_box()**, 563–564  
**put\_on\_top()**, 1211  
**show()**, 562  
technical example, 1213–12116  
text input/output, 563–564  
visibility, 562  
Wild cards, regular expressions, 1178  
Wilkes, Maurice, 820  
**Window** example, 420, 443  
canvas, 420  
creating, 422–424, 554–556  
disappearing, 576  
drawing area, 420  
implementation, 1210–1212

line drawing example, 565–569  
**put\_on\_top()**, 1211  
**Window.h** example, 421–422  
Wirth, Niklaus, 830–831  
Word frequency, example, 777  
Words (of memory), 1222  
**write()**, unformatted output, 1173  
Writing files, 350. *See also* File I/O  
    appending to, 389  
    binary I/O, 391  
    example, 352–354  
    **fstream** type, 350–352  
    **ofstream** type, 351–352  
    **ostream** type, 349–354, 391  
**ws** manipulator, 1174

## X

**xdigit**, 878, 1179  
**\xhhh**, hexadecimal character literal, 1080  
**xor**, synonym for **^**, 1038  
**xor\_eq**, synonym for **^=**, 1038

## Z

zero-terminated array, 1045. *See also* C-style strings  
ZIP code example, 880–885