

Programming

Second Edition



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 or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.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.
First printing, May 2014

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	vector	117
4.6.1	Traversing a vector	119
4.6.2	Growing a vector	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>%</code> 230	
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- const -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	constexpr 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	using declarations and using directives	294

Chapter 9 Technicalities: Classes, etc. 303

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	struct 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	const member functions	330
9.7.5	Members and “helper functions”	332
9.8	The Date 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	Rectangle	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 **Matrix** library 897
 24.5.1 Dimensions and access 898
 24.5.2 1D **Matrix** 901
 24.5.3 2D **Matrix** 904
 24.5.4 **Matrix** I/O 907
 24.5.5 3D **Matrix** 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 **bitset** 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	struct 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	sizeof	1093
A.5.5	Logical expressions	1094
A.5.6	new and delete	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	#include	1128
A.17.2	#define	1129

Appendix B Standard Library Summary 1131

B.1	Overview	1132
B.1.1	Header files	1133
B.1.2	Namespace std	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	min and max	1161
B.6	STL utilities	1162
B.6.1	Inserters	1162
B.6.2	Function objects	1163
B.6.3	pair and tuple	1165
B.6.4	initializer_list	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	valarray	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 printf() family	1186
B.11.3	C-style strings	1191
B.11.4	Memory	1192
B.11.5	Date and time	1193
B.11.6	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 std_lib_facilities.h 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

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. 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, 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 Daugherity, 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 Versoondor, 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 McLaren 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.



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	0.4 Creativity and problem solving
0.1.1 General approach	
0.1.2 Drills, exercises, etc.	0.5 Request for feedback
0.1.3 What comes after this book?	0.6 References
0.2 A philosophy of teaching and learning	0.7 Biographies
0.2.1 The order of topics	
0.2.2 Programming and programming language	
0.2.3 Portability	
0.3 Programming and computer science	

0.1 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.

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.

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. 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.



Computers, People, and Programming

“Specialization is for insects.”

—R. A. Heinlein

In this chapter, we present some of the things that we think make programming important, interesting, and fun. We also present a few fundamental ideas and ideals. We hope to debunk a couple of popular myths about programming and programmers. This is a chapter to skim for now and to return to later when you are struggling with some programming problem and wondering if it's all worth it.

1.1 Introduction**1.2 Software****1.3 People****1.4 Computer science****1.5 Computers are everywhere**

- 1.5.1 Screens and no screens
- 1.5.2 Shipping
- 1.5.3 Telecommunications
- 1.5.4 Medicine
- 1.5.5 Information
- 1.5.6 A vertical view
- 1.5.7 So what?

1.6 Ideals for programmers

1.1 Introduction

Like most learning, learning how to program is a chicken and egg problem: We want to get started, but we also want to know why what we are about to learn matters. We want to learn a practical skill, but also make sure it is not just a passing fad. We want to know that we are not going to waste our time, but don't want to be bored by still more hype and moralizing. For now, just read as much of this chapter as seems interesting and come back later when you feel the need to refresh your memory of why the technical details matter outside the classroom.

This chapter is a personal statement of what we find interesting and important about programming. It explains what motivates us to keep going in this field after decades. This is a chapter to read to get an idea of possible ultimate goals and an idea of what kind of person a programmer might be. A beginner's technical book inevitably contains much pretty basic stuff. In this chapter, we lift our eyes from the technical details and consider the big picture: Why is programming a worthwhile activity? What is the role of programming in our civilization? Where can a programmer make contributions to be proud of? Where does programming fit into the greater world of software development, deployment, and maintenance? When people talk about “computer science,” “software engineering,” “information technology,” etc., where does programming fit into the picture? What does a programmer do? What skills does a good programmer have?

To a student, the most urgent reason for understanding an idea, a technique, or a chapter may be to pass a test with a good grade – but there has to be more to learning than that! To someone working in the software industry, the most urgent

reason for understanding an idea, a technique, or a chapter may be to find something that can help with the current project and that will not annoy the boss who controls the next paycheck, promotions, and firings – but there has to be more to learning than that! We work best when we feel that our work in some small way makes the world a better place for people to live in. For tasks that we perform over a period of years (the “things” that professions and careers are made of), ideals and more abstract ideas are crucial.

Our civilization runs on software. Improving software and finding new uses for software are two of the ways an individual can help improve the lives of many. Programming plays an essential role in that.



1.2 Software

Good software is invisible. You can't see it, feel it, weigh it, or knock on it. *Software* is a collection of programs running on some computer. Sometimes, we can see the computer. Often, we can see only something that contains the computer, such as a telephone, a camera, a bread maker, a car, or a wind turbine. We can see what that software does. We can be annoyed or hurt if it doesn't do what it is supposed to do. We can be annoyed or hurt if what it is supposed to do doesn't suit our needs.

How many computers are there in the world? We don't know; billions at least. There may be more computers in the world than people. We need to count servers, desktop computers, laptops, tablets, smartphones, and computers embedded in “gadgets.”

How many computers do you (more or less directly) use every day? There are more than 30 computers in my car, two in my cell phone, one in my MP3 player, and one in my camera. Then there is my laptop (on which the page you are reading is being written) and my desktop machine. The air-conditioning controller that keeps the summer heat and humidity at bay is a simple computer. There is one controlling the computer science department's elevator. If you use a modern television, there will be at least one computer in there somewhere. A bit of web surfing gets you into direct contact with dozens – possibly hundreds – of servers through a telecommunications system consisting of many thousands of computers – telephone switches, routers, and so on.

No, I do not drive around with 30 laptops on the backseat of my car! The point is that most computers do not look like the popular image of a computer (with a screen, a keyboard, a mouse, etc.); they are small “parts” embedded in the equipment we use. So, that car has nothing that looks like a computer, not even a screen to display maps and driving directions (though such gadgets are popular in other cars). However, its engine contains quite a few computers, doing things like fuel injection control and temperature monitoring. The power-assisted steering involves at least one computer, the radio and the security

system contain some, and we suspect that even the open/close controls of the windows are computer controlled. Newer models even have computers that continuously monitor tire pressure.

How many computers do you depend on for what you do during a day? You eat; if you live in a modern city, getting the food to you is a major effort requiring minor miracles of planning, transport, and storage. The management of the distribution networks is of course computerized, as are the communication systems that stitch them all together. Modern farming is highly computerized; next to the cow barn you find computers used to monitor the herd (ages, health, milk production, etc.), farm equipment is increasingly computerized, and the number of forms required by the various branches of government can make any honest farmer cry. If something goes wrong, you can read all about it in your newspaper; of course, the articles in that paper were written on computers, set on the page by computers, and (if you still read the “dead tree edition”) printed by computerized equipment – often after having been electronically transmitted to the printing plant. Books are produced in the same way. If you have to commute, the traffic flows are monitored by computers in a (usually vain) attempt to avoid traffic jams. You prefer to take the train? That train will also be computerized; some even operate without a driver, and the train’s subsystems, such as announcements, braking, and ticketing, involve lots of computers. Today’s entertainment industry (music, movies, television, stage shows) is among the largest users of computers. Even non-cartoon movies use (computer) animation heavily; music and photography are also digital (i.e., using computers) for both recording and delivery. Should you become ill, the tests your doctor orders will involve computers, the medical records are often computerized, and most of the medical equipment you’ll encounter if you are sent to a hospital to be cured contains computers. Unless you happen to be staying in a cottage in the woods without access to any electrically powered gadgets (including light bulbs), you use energy. Oil is found, extracted, processed, and distributed through a system using computers every step along the way, from the drill bit deep in the ground to your local gas (petrol) pump. If you pay for that gas with a credit card, you again exercise a whole host of computers. It is the same story for coal, gas, solar, and wind power.

The examples so far are all “operational”; they are directly involved in what you are doing. Once removed from that is the important and interesting area of design. The clothes you wear, the telephone you talk into, and the coffee machine that dispenses your favorite brew were designed and manufactured using computers. The superior quality of modern photographic lenses and the exquisite shapes in the design of modern everyday gadgets and utensils owe almost everything to computer-based design and production methods. The craftsmen/designers/artists/engineers who design our environment have been freed from many physical con-

straints previously considered fundamental. If you get ill, the medicines given to cure you will have been designed using computers.

Finally, research – science itself – relies heavily on computers. The telescopes that probe the secrets of distant stars could not be designed, built, or operated without computers, and the masses of data they produce couldn't be analyzed and understood without computers. An individual biology field researcher may not be heavily computerized (unless, of course, a camera, a digital tape recorder, a telephone, etc. are used), but back in the lab, the data has to be stored, analyzed, checked against computer models, and communicated to fellow scientists. Modern chemistry and biology – including medical research – use computers to an extent undreamed of a few years ago and still unimagined by most people. The human genome was sequenced by computers. Or – let's be precise – the human genome was sequenced by humans using computers. In all of these examples, we see computers as something that enables us to do something we would have had a harder time doing without computers.

Every one of those computers runs software. Without software, they would just be expensive lumps of silicon, metal, and plastic: doorstops, boat anchors, and space heaters. Every line of that software was written by some individual. Every one of those lines that was actually executed was minimally reasonable, if not correct. It's amazing that it all works! We are talking about billions of lines of code (program text) in hundreds of programming languages. Getting all that to work took a staggering amount of effort and involved an unimaginable number of skills. We want further improvements to essentially every service and gadget we depend on. Just think of any one service and gadget you rely on; what would you like to see improved? If nothing else, we want our services and gadgets smaller (or bigger), faster, more reliable, with more features, easier to use, with higher capacity, better looking, and cheaper. The likelihood is that the improvement you thought of requires some programming.

1.3 People

Computers are built by people for the use of people. A computer is a very generic tool; it can be used for an unimaginable range of tasks. It takes a program to make it useful to someone. In other words, a computer is just a piece of hardware until someone – some programmer – writes code for it to do something useful. We often forget about the software. Even more often, we forget about the programmer.

Hollywood and similar “popular culture” sources of disinformation have assigned largely negative images to programmers. For example, we have all seen the solitary, fat, ugly nerd with no social skills who is obsessed with video games and breaking into other people's computers. He (almost always a male) is as likely to



want to destroy the world as he is to want to save it. Obviously, milder versions of such caricatures exist in real life, but in our experience they are no more frequent among software developers than they are among lawyers, police officers, car salesmen, journalists, artists, or politicians.

Think about the applications of computers you know from your own life. Were they done by a loner in a dark room? Of course not; the creation of a successful piece of software, computerized gadget, or system involves dozens, hundreds, or thousands of people performing a bewildering set of roles: for example, programmers, (program) designers, testers, animators, focus group managers, experimental psychologists, user interface designers, analysts, system administrators, customer relations people, sound engineers, project managers, quality engineers, statisticians, hardware interface engineers, requirements engineers, safety officers, mathematicians, sales support personnel, troubleshooters, network designers, methodologists, software tools managers, software librarians, etc. The range of roles is huge and made even more bewildering by the titles varying from organization to organization: one organization’s “engineer” may be another organization’s “programmer” and yet another organization’s “developer,” “member of technical staff,” or “architect.” There are even organizations that let their employees pick their own titles. Not all of these roles directly involve programming. However, we have personally seen examples of people performing each of the roles mentioned while reading or writing code as an essential part of their job. Additionally, a programmer (performing any of these roles, and more) may over a short period of time interact with a wide range of people from application areas, such as biologists, engine designers, lawyers, car salesmen, medical researchers, historians, geologists, astronauts, airplane engineers, lumberyard managers, rocket scientists, bowling alley builders, journalists, and animators (yes, this is a list drawn from personal experience). Someone may also be a programmer at times and fill non-programming roles at other stages of a professional career.

The myth of a programmer being isolated is just that: a myth. People who like to work on their own choose areas of work where that is most feasible and usually complain bitterly about the number of “interruptions” and meetings. People who prefer to interact with other people have an easier time because modern software development is a team activity. The implication is that social and communication skills are essential and valued far more than the stereotypes indicate. On a short list of highly desirable skills for a programmer (however you realistically define *programmer*), you find the ability to communicate well – with people from a wide variety of backgrounds – informally, in meetings, in writing, and in formal presentations. We are convinced that until you have completed a team project or two, you have no idea of what programming is and whether you really like it. Among the things we like about programming are all the nice and

interesting people we meet and the variety of places we get to visit as part of our professional lives.

One implication of all this is that people with a wide variety of skills, interests, and work habits are essential for producing good software. Our quality of life depends on those people – sometimes even our life itself. No one person could fill all the roles we mention here; no sensible person would want every role. The point is that you have a wider choice than you could possibly imagine; not that you have to make any particular choice. As an individual you will “drift” toward areas of work that match your skills, talents, and interests.

We talk about “programmers” and “programming,” but obviously programming is only part of the overall picture. The people who design a ship or a cell phone don’t think of themselves as programmers. Programming is an important part of software development, but not all there is to software development. Similarly, for most products, software development is an important part of product development, but not all there is to product development.

We do not assume that you – our reader – want to become a professional programmer and spend the rest of your working life writing code. Even the best programmers – especially the *best* programmers – spend most of their time *not* writing code. Understanding problems takes serious time and often requires significant intellectual effort. That intellectual challenge is what many programmers refer to when they say that programming is interesting. Many of the best programmers also have degrees in subjects not usually considered part of computer science. For example, if you work on software for genomic research, you will be much more effective if you understand some molecular biology. If you work on programs for analyzing medieval literature, you could be much better off reading a bit of that literature and maybe even knowing one or more of the relevant languages. In particular, a person with an “all I care about is computers and programming” attitude will be incapable of interacting with his or her non-programmer colleagues. Such a person will not only miss out on the best parts of human interactions (i.e., life) but also be a bad software developer.

So, what do we assume? Programming is an intellectually challenging set of skills that are part of many important and interesting technical disciplines. In addition, programming is an essential part of our world, so not knowing the basics of programming is like not knowing the basics of physics, history, biology, or literature. Someone totally ignorant of programming is reduced to believing in magic and is dangerous in many technical roles. If you read Dilbert, think of the pointy-haired boss as the kind of manager you don’t want to meet or (far worse) become. In addition, programming can be fun.

But what do we assume you might use programming for? Maybe you will use programming as a key tool in your further studies and work without becoming a professional programmer. Maybe you will interact with other people

professionally and personally in ways where a basic knowledge of programming will be an advantage, maybe as a designer, writer, manager, or scientist. Maybe you will do programming at a professional level as part of your studies or work. Even if you do become a professional programmer it is unlikely that you will do nothing but programming.

You might become an engineer focusing on computers or a computer scientist, but even then you will not “program all the time.” Programming is a way of presenting ideas in code – a way of aiding problem solving. It is nothing – absolutely a waste of time – unless you have ideas that are worth presenting and problems worth solving.

This is a book about programming and we have promised to help you learn how to program, so why do we emphasize non-programming subjects and the limited role of programming? A good programmer understands the role of code and programming technique in a project. A good programmer is (at most times) a good team player and tries hard to understand how the code and its production best support the overall project. For example, imagine that I worked on a new MP3 player (maybe to be part of a smartphone or a tablet) and all that I cared about was the beauty of my code and the number of neat features I could provide. I would probably insist on the largest, most powerful computer to run my code. I might disdain the theory of sound encoding because it is “not programming.” I would stay in my lab, rather than go out to meet potential users, who undoubtedly would have bad tastes in music anyway and would not appreciate the latest advances in GUI (graphical user interface) programming. The likely result would be disaster for the project. A bigger computer would mean a costlier MP3 player and most likely a shorter battery life. Encoding is an essential part of handling music digitally, so failing to pay attention to advances in encoding techniques could lead to increased memory requirements for each song (encodings differ by as much as 100% for the same-quality output). A disregard for users’ preferences – however odd and archaic they may seem to you – typically leads to the users choosing some other product. An essential part of writing a good program is to understand the needs of the users and the constraints that those needs place on the implementation (i.e., the code). To complete this caricature of a bad programmer, we just have to add a tendency to deliver late because of an obsession with details and an excessive confidence in the correctness of lightly tested code. We encourage you to become a good programmer, with a broad view of what it takes to produce good software. That’s where both the value to society and the keys to personal satisfaction lie.

1.4 Computer science

Even by the broadest definition, programming is best seen as a part of something greater. We can see it as a subdiscipline of computer science, computer engineering, software engineering, information technology, or any other software-related disci-

pline. We see programming as an enabling technology for those computer and information fields of science and engineering, as well as for physics, biology, medicine, history, literature, and any other academic or research field.

Consider computer science. A 1995 U.S. government “blue book” defines it like this: “The systematic study of computing systems and computation. The body of knowledge resulting from this discipline contains theories for understanding computing systems and methods; design methodology, algorithms, and tools; methods for the testing of concepts; methods of analysis and verification; and knowledge representation and implementation.” As we would expect, the Wikipedia entry is less formal: “Computer science, or computing science, is the study of the theoretical foundations of information and computation and their implementation and application in computer systems. Computer science has many sub-fields; some emphasize the computation of specific results (such as computer graphics), while others (such as computational complexity theory) relate to properties of computational problems. Still others focus on the challenges in implementing computations. For example, programming language theory studies approaches to describing computations, while computer programming applies specific programming languages to solve specific computational problems.”

Programming is a tool; it is a fundamental tool for expressing solutions to fundamental and practical problems so that they can be tested, improved through experiment, and used. Programming is where ideas and theories meet reality. This is where computer science can become an experimental discipline, rather than pure theory, and impact the world. In this context, as in many others, it is essential that programming is an expression of well-tried practices as well as the theories. It must not degenerate into mere hacking: just get some code written, any old way that meets an immediate need.

1.5 Computers are everywhere

Nobody knows everything there is to know about computers or software. This section just gives you a few examples. Maybe you’ll see something you like. At least you might be convinced that the scope of computer use – and through that, programming – is far larger than any individual can fully grasp.

Most people think of a computer as a small gray box attached to a screen and a keyboard. Such computers tend to be good at games, messaging and email, and playing music. Other computers, called laptops, are used on planes by bored businessmen to look at spreadsheets, play games, and watch videos. This caricature is just the tip of the iceberg. Most computers work out of our sight and are part of the systems that keep our civilization going. Some fill rooms; others are smaller than a small coin. Many of the most interesting computers don’t directly interact with a human through a keyboard, mouse, or similar gadget.

1.5.1 Screens and no screens

The idea of a computer as a fairly large rectangular box with a screen and a keyboard is common and often hard to shake off. However, consider these two computers:



Both of these “gadgets” (which happen to be watches) are primarily computers. In fact, we conjecture that they are essentially the same model computer with different I/O (input/output) systems. The left one drives a small screen (similar to the screens on conventional computers, but smaller) and the second drives little electric motors controlling traditional clock hands and a disk of numbers for day-of-month readout. Their input systems are the four buttons (more easily seen on the right-hand watch) and a radio receiver, used for synchronization with very high-precision “atomic” clocks. Most of the programs controlling these two computers are shared between them.

1.5.2 Shipping

These two photos show a large marine diesel engine and the kind of huge ship that it may power:



Consider where computers and software play key roles here:

- *Design:* Of course, the ship and the engine were both designed using computers. The list of uses is almost endless and includes architectural and engineering drawings, general calculations, visualization of spaces and parts, and simulations of the performance of parts.
- *Construction:* A modern shipyard is heavily computerized. The assembly of a ship is carefully planned using computers, and the work is guided by computers. Welding is done by robots. In particular, a modern double-hulled tanker couldn't be built without little welding robots to do the welding from within the space between the hulls. There just isn't room for a human in there. Cutting steel plates for a ship was one of the world's first CAD/CAM (computer-aided design and computer-aided manufacture) applications.
- *The engine:* The engine has electronic fuel injection and is controlled by a few dozen computers. For a 100,000-horsepower engine (like the one in the photo), that's a nontrivial task. For example, the engine management computers continuously adjust fuel mix to minimize the pollution that would result from a badly tuned engine. Many of the pumps associated with the engine (and other parts of the ship) are themselves computerized.
- *Management:* Ships sail where there is cargo to pick up and to deliver. The scheduling of fleets of ships is a continuing process (computerized, of course) so that routings change with the weather, with supply and demand, and with space and loading capacity of harbors. There are even websites where you can watch the position of major merchant vessels at any time. The ship in the photo happens to be a container vessel (one of the largest such in the world; 397m long and 56m wide), but other kinds of large modern ships are managed in similar ways.
- *Monitoring:* An oceangoing ship is largely autonomous; that is, its crew can handle most contingencies likely to arise before the next port. However, they are also part of a globe-spanning network. The crew has access to reasonably accurate weather information (from and through – computerized – satellites). They have a GPS (global positioning system) and computer-controlled and computer-enhanced radar. If the crew needs a rest, most systems (including the engine, radar, etc.) can be monitored (via satellite) from a shipping-line control room. If anything unusual is spotted, or if the connection "back home" is broken, the crew is notified.

Consider the implication of a failure of one of the hundreds of computers explicitly mentioned or implied in this brief description. Chapter 25 ("Embedded Systems Programming") examines this in slightly more detail. Writing code for a modern ship is a skilled and interesting activity. It is also useful. The cost of

sea transport is really amazingly low. You appreciate that when you buy something that wasn't manufactured locally. Sea transport has always been cheaper than land transport; these days one of the reasons is serious use of computers and information.

1.5.3 Telecommunications

These two photos show a telephone switch and a telephone (that also happens to be a camera, an MP3 player, an FM radio, a web browser, and much more):



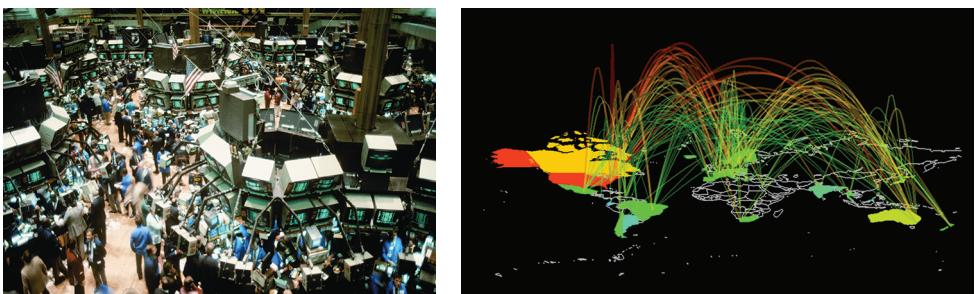
Consider where computers and software play key roles here. You pick up a telephone and “dial,” the person you dialed answers, and you talk. Or maybe you get to leave a voicemail, or maybe you send a photo from your phone camera, or maybe you send a text message (hit Send and let the phone do the dialing). Obviously the phone is a computer. This is especially obvious if the phone (like most mobile phones) has a screen and allows more than traditional “plain old telephone services,” such as web browsing. Actually, such phones tend to contain several computers: one to manage the screen, one to talk to the phone system, and maybe more.

The part of the phone that manages the screen, does web browsing, etc. is probably the most familiar to computer users: it just runs a graphical user interface to “all the usual stuff.” What is unknown to and largely unsuspected by most users is the huge system that the little phone talks to while doing its job. I dial a number in Texas, but you are on vacation in New York City, yet within seconds your phone rings and I hear your “Hello!” over the roar of city traffic. Many phones can perform that trick for essentially any two locations on earth and we just take it for granted. How did my phone find yours? How is the sound transmitted? How is the sound encoded into data packets? The answer could fill

many books much thicker than this one, but it involves a combination of hardware and software on hundreds of computers scattered over the geographical area in question. If you are unlucky, a few telecommunications satellites (themselves computerized systems) are also involved – “unlucky” because we cannot perfectly compensate for the 20,000-mile detour out into space; the speed of light (and therefore the speed of your voice) is finite (light fiber cables are much better: shorter, faster, and carrying much more data). Most of this works remarkably well; the backbone telecommunications systems are 99.9999% reliable (for example, 20 minutes of downtime in 20 years – that’s $20/20 \times 365 \times 24 \times 60$). The trouble we have tends to be in the communications between our mobile phone and the nearest main telephone switch.

There is software for connecting the phones, for chopping our spoken words into data packets to be sent over wires and radio links, for routing those messages, for recovering from all kinds of failures, for continuously monitoring the quality and reliability of the services, and of course for billing. Even keeping track of all the physical pieces of the system requires serious amounts of clever software: What talks to what? What parts go into a new system? When do you need to do some preventive maintenance?

Arguably the backbone telecommunications system of the world, consisting of semi-independent but interconnected systems, is the largest and most complicated man-made artifact. To make things a bit more real: remember, this is not just boring old telephony with a few new bells and whistles. The various infrastructures have merged. They are also what the internet (the web) runs on, what our banking and trading systems run on, and what carry our television programs to the broadcasting stations. So, we can add another couple of photos to illustrate telecommunications:



The room is the “trading floor” of the American stock exchange on New York’s Wall Street and the map is a representation of parts of the internet backbones (a complete map would be too messy to be useful).

As it happens, we also like digital photography and the use of computers to draw specialized maps to visualize knowledge.

1.5.4 Medicine

These two photos show a CAT (computed axial tomography) scanner and an operating theater for computer-aided surgery (also called “robot-assisted surgery” or “robotic surgery”):

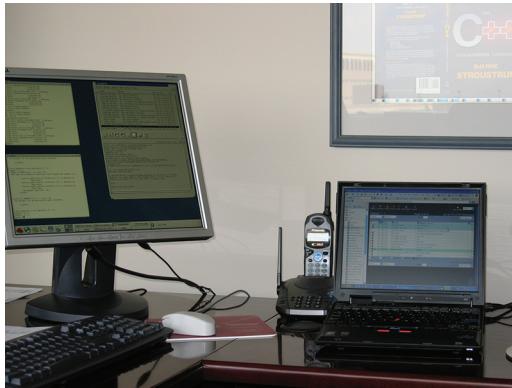


Consider where computers and software play key roles here. The scanners basically are computers; the pulses they send out are controlled by a computer, and the readings are nothing but gibberish until quite sophisticated algorithms are applied to convert them to something we recognize as a (three-dimensional) image of the relevant part of a human body. To do computerized surgery, we must go several steps further. A wide variety of imaging techniques are used to let the surgeon see the inside of the patient, to see the point of surgery with significant enlargement or in better light than would otherwise be possible. With the aid of a computer a surgeon can use tools that are too fine for a human hand to hold or in a place where a human hand could not reach without unnecessary cutting. The use of minimally invasive surgery (laparoscopic surgery) is a simple example of this that has minimized the pain and recovery time for millions of people. The computer can also help steady the surgeon’s “hand” to allow for more delicate work than would otherwise be possible. Finally, a “robotic” system can be operated remotely, thus making it possible for a doctor to help someone remotely (over the internet). The computers and programming involved are mind-boggling, complex, and interesting. The user interface, equipment control, and imaging challenges alone will keep thousands of researchers, engineers, and programmers busy for decades.

We heard of a discussion among a large group of medical doctors about which new tool had provided the most help to them in their work: The CAT scanner? The MRI scanner? The automated blood analysis machines? The high-resolution ultrasound machines? PDAs? After some discussion, a surprising “winner” of this “competition” emerged: instant access to patient records. Knowing the medical history of a patient (earlier illnesses, medicines tried earlier, allergies, hereditary problems, general health, current medication, etc.) simplifies the problem of diagnosis and minimizes the chance of mistakes.

1.5.5 Information

These two photos show an ordinary PC (well, two) and part of a server farm:



We have focused on “gadgets” for the usual reason: you cannot see, feel, or hear software. We cannot present you with a photograph of a neat program, so we show you a “gadget” that runs one. However, much software deals directly with “information.” So let’s consider “ordinary uses” of “ordinary computers” running “ordinary software.”

A “server farm” is a collection of computers providing web services. Organizations running state-of-the-art server farms (such as Google, Amazon, and Microsoft) are somewhat close-mouthed about the details of their servers, and the specifications of server farms change constantly (so most of the information you find on the web is outdated). However, the specifications are amazing and should convince anyone that there is more to programming than simply computing a few numbers on a laptop:

- Google uses about a million servers (each more powerful than your laptop) in 25 to 50 “data centers.”
- Such a data center is housed in a warehouse that might measure 60m*100m (that’s about 200ft*330ft) or more.
- In 2011, the *New York Times* reported that Google’s data centers draw about 260 million watts continuously (about the same amount of energy as Las Vegas).
- Assume a server machine to be a 3GHz quad-core with 24GB of main memory. That would imply about 12×10^{15} Hz of compute power (about 12,000,000,000,000 instructions per second) with 24×10^{15} bytes of main memory (about 24,000,000,000,000,000 8-bit bytes), and maybe 4TB of disk per server, giving 4×10^{18} bytes of storage.

We may be underestimating the amounts, and by the time you read this, we almost certainly are. In particular, efforts to minimize energy usage seem to be driving machine architectures toward more processors per server and more cores per processor. A GB is a gigabyte, that is, about 10^9 characters. A TB, a terabyte, is about 1000GB, that is, about 10^{12} characters. A PB, a petabyte (that is, 10^{15} bytes), is becoming a more common measure. This is a pretty extreme example, but every major company runs programs on the web to interact with its users/customers. Examples are Amazon (book and other sales), Amadeus (airline ticketing and automobile rental), and eBay (online auctions). Millions of companies, organizations, and individuals also have a presence on the web. Most don't run their own software, but many do and much of that is not trivial.

The other, and more traditional, massive computing effort involves accounting, order processing, payroll, record keeping, billing, inventory management, personnel records, student records, patient records, etc. – the records that essentially every organization (commercial and noncommercial, governmental and private) keeps. These records are the backbone of their respective organizations. As a computing effort, processing such records seems simple: mostly some information (records) is just stored and retrieved and very little is done to it. Examples include

- Is my 12:30 flight to Chicago still on time?
- Has Gilbert Sullivan had the measles?
- Has the coffeemaker that Juan Valdez ordered been shipped?
- What kind of kitchen chair did Jack Sprat buy in 1996 (or so)?
- How many phone calls originated from the 212 area code in August of 2012?
- What was the number of coffeepots sold in January and for what total price?

The sheer scale of the databases involved makes these systems highly complex. To that add the need to respond quickly (often in less than two seconds for individual queries) and to be correct (at least most of the time). These days, it is not uncommon for people to talk about terabytes of data (a byte is the amount of memory needed to hold an ordinary character). That's traditional "data processing" and it is merging with "the web" because most access to the databases is now through web interfaces.

This kind of computer use is often referred to as *information processing*. It focuses on data – often lots of data. This leads to challenges in the organization and transmission of data and lots of interesting work on how to present vast amounts of data in a comprehensible form: "user interface" is a very important aspect of handling data. For example, think of analyzing a work of older literature (say, Chaucer's *Canterbury Tales* or Cervantes' *Don Quixote*) to figure out what the author

actually wrote by comparing dozens of versions. We need to search through the texts with a variety of criteria supplied by the person doing the analysis and to display the results in a way that aids the discovery of salient points. Thinking of text analysis, publishing comes to mind: today, just about every article, book, brochure, newspaper, etc. is produced on a computer. Designing software to support that well is for most people still a problem that lacks a really good solution.

1.5.6 A vertical view

It is sometimes claimed that a paleontologist can reconstruct a complete dinosaur and describe its lifestyle and natural environment from studying a single small bone. That may be an exaggeration, but there is something to the idea of looking at a simple artifact and thinking about what it implies. Consider this photo showing the landscape of Mars taken by a camera on one of NASA's Mars Rovers:



If you want to do “rocket science,” becoming a good programmer is one way. The various space programs employ lots of software designers, especially ones who can also understand some of the physics, math, electrical engineering, mechanical engineering, medical engineering, etc. that underlie the manned and unmanned space programs. Getting those two Rovers to drive around on Mars for years is one of the greatest technological triumphs of our civilization. One (*Spirit*) sent data back for six years and the other (*Opportunity*) is still working at the time of writing and will have its tenth anniversary on Mars in January 2014. Their estimated design life was three months.

The photo was transmitted to earth through a communication channel with a 25-minute transmission delay each way; there is a lot of clever programming and advanced math to make sure that the picture is transmitted using the minimal number of bits without losing any of them. On earth, the photo is then rendered using algorithms to restore color and minimize distortion due to the optics and electronic sensors.

The control programs for the Mars Rovers are of course programs – the Rovers drive autonomously for 24 hours at a time and follow instructions sent from earth the day before. The transmission is managed by programs.

The operating systems used for the various computers involved in the Rovers, the transmission, and the photo reconstruction are programs, as are the applications used to write this chapter. The computers on which these programs run are designed and produced using CAD/CAM (computer-aided design and computer-aided manufacture) programs. The chips that go into those computers are produced on computerized assembly lines constructed using precision tools, and those tools also use computers (and software) in their design and manufacture. The quality control for those long construction processes involves serious computation. All that code was written by humans in a high-level programming language and translated into machine code by a compiler, which is itself such a program. Many of these programs interact with users using GUIs and exchange data using input/output streams.

Finally, a lot of programming goes into image processing (including the processing of the photos from the Mars Rovers), animation, and photo editing (there are versions of the Rover photos floating around on the web featuring “Martians”).

1.5.7 So what?

What do all these “fancy and complicated” applications and software systems have to do with learning programming and using C++? The connection is simply that many programmers do get to work on projects like these. These are the kinds of things that good programming can help achieve. Also, every example used in this chapter involved C++ and at least some of the techniques we describe in this book. Yes, there are C++ programs in MP3 players, in ships, in wind turbines, on Mars, and in the human genome project. For more applications using C++, see www.stroustrup.com/applications.html.

1.6 Ideals for programmers

What do we want from our programs? What do we want in general, as opposed to a particular feature of a particular program? We want *correctness* and as part of that, *reliability*. If the program doesn’t do what it is supposed to do, and do so in a way so that we can rely on it, it is at best a serious nuisance, at worst a danger. We want it to be *well designed* so that it addresses a real need well; it doesn’t really matter that a program is correct if what it does is irrelevant to us or if it correctly does something in a way that annoys us. We also want it to be *affordable*; I might prefer a Rolls-Royce or an executive jet to my usual forms of transport, but unless I’m a zillionaire, cost will enter into my choices.

These are aspects of software (gadgets, systems) that can be appreciated from the outside, by non-programmers. They must be ideals for programmers and we must keep them in mind at all times, especially in the early phases of development,

if we want to produce successful software. In addition, we must concern ourselves with ideals related to the code itself: our code must be *Maintainable*; that is, its structure must be such that someone who didn't write it can understand it and make changes. A successful program "lives" for a long time (often for decades) and will be changed again and again. For example, it will be moved to new hardware, it will have new features added, it will be modified to use new I/O facilities (screens, video, sound), to interact using new natural languages, etc. Only a failed program will never be modified. To be maintainable, a program must be simple relative to its requirements, and the code must directly represent the ideas expressed. Complexity – the enemy of simplicity and maintainability – can be intrinsic to a problem (in that case we just have to deal with it), but it can also arise from poor expression of ideas in code. We must try to avoid that through good coding style – style matters!

This doesn't sound too difficult, but it is. Why? Programming is fundamentally simple: just tell the machine what it is supposed to do. So why can programming be most challenging? Computers are fundamentally simple; they can just do a few operations, such as adding two numbers and choosing the next instruction to execute based on a comparison of two numbers. The problem is that we don't want computers to do simple things. We want "the machine" to do things that are difficult enough for us to want help with them, but computers are nitpicking, unforgiving, dumb beasts. Furthermore, the world is more complex than we'd like to believe, so we don't really know the implications of what we request. We just want a program to "do something like this" and don't want to be bothered with technical details. We also tend to assume "common sense." Unfortunately, common sense isn't all that common among humans and is totally absent in computers (though some really well-designed programs can imitate it in specific, well-understood cases).

This line of thinking leads to the idea that "programming is understanding": when you can program a task, you understand it. Conversely, when you understand a task thoroughly, you can write a program to do it. In other words, we can see programming as part of an effort to thoroughly understand a topic. A program is a precise representation of our understanding of a topic.

When you program, you spend significant time trying to understand the task you are trying to automate.

We can describe the process of developing a program as having four stages:

- *Analysis*: What's the problem? What does the user want? What does the user need? What can the user afford? What kind of reliability do we need?
- *Design*: How do we solve the problem? What should be the overall structure of the system? Which parts does it consist of? How do those parts communicate with each other? How does the system communicate with its users?

- *Programming:* Express the solution to the problem (the design) in code. Write the code in a way that meets all constraints (time, space, money, reliability, and so on). Make sure that the code is correct and maintainable.
- *Testing:* Make sure the system works correctly under all circumstances required by systematically trying it out.

Programming plus testing is often called *implementation*. Obviously, this simple split of software development into four parts is a simplification. Thick books have been written on each of these four topics and more books still about how they relate to each other. One important thing to note is that these stages of development are not independent and do not occur strictly in sequence. We typically start with analysis, but feedback from testing can help improve the programming; problems with getting the program working may indicate a problem with the design; and working with the design may suggest aspects of the problem that hitherto had been overlooked in the analysis. Actually using the system typically exposes weaknesses of the analysis.



The crucial concept here is *feedback*. We learn from experience and modify our behavior based on what we learn. That's essential for effective software development. For any large project, we don't know everything there is to know about the problem and its solution before we start. We can try out ideas and get feedback by programming, but in the earlier stages of development it is easier (and faster) to get feedback by writing down design ideas, trying out those design ideas, and using scenarios on friends. The best design tool we know of is a blackboard (use a whiteboard instead if you prefer chemical smells over chalk dust). Never design alone if you can avoid it! Don't start coding before you have tried out your ideas by explaining them to someone. Discuss designs and programming techniques with friends, colleagues, potential users, and so on before you head for the keyboard. It is amazing how much you can learn from simply trying to articulate an idea. After all, a program is nothing more than an expression (in code) of some ideas.

Similarly, when you get stuck implementing a program, look up from the keyboard. Think about the problem itself, rather than your incomplete solution. Talk with someone: explain what you want to do and why it doesn't work. It's amazing how often you find the solution just by carefully explaining the problem to someone. Don't debug (find program errors) alone if you don't have to!

The focus of this book is implementation, and especially programming. We do not teach "problem solving" beyond giving you plenty of examples of problems and their solutions. Much of problem solving is recognizing a known problem and applying a known solution technique. Only when most subproblems are handled this way will you find the time to indulge in exciting and creative "out-of-the-box thinking." So, we focus on showing how to express ideas clearly in code.

Direct expression of ideas in code is a fundamental ideal of programming. That's really pretty obvious, but so far we are a bit short of good examples. We'll come back to this, repeatedly. When we want an integer in our code, we store it in an **int**, which provides the basic integer operations. When we want a string of characters, we store it in a **string**, which provides the most basic text manipulation operations. At the most fundamental level, the ideal is that when we have an idea, a concept, an entity, something we think of as a "thing," something we can draw on our whiteboard, something we can refer to in our discussions, something our (non-computer science) textbook talks about, then we want that something to exist in our program as a named entity (a type) providing the operations we think appropriate for it. If we want to do math, we want a **complex** type for complex numbers and a **Matrix** type for linear algebra. If we want to do graphics, we want a **Shape** type, a **Circle** type, a **Color** type, and a **Dialog_box**. When we want to deal with streams of data, say from a temperature sensor, we want an **istream** type (**i** for input). Obviously, every such type should provide the appropriate operations and only the appropriate operations. These are just a few examples from this book. Beyond that, we offer tools and techniques for you to build your own types to directly represent whatever concepts you want in your program.

Programming is part practical, part theoretical. If you are just practical, you will produce non-scalable, unmaintainable hacks. If you are just theoretical, you will produce unusable (or unaffordable) toys.

For a different kind of view of the ideals of programming and a few people who have contributed in major ways to software through work with programming languages, see Chapter 22, "Ideals and History."

Review

Review questions are intended to point you to the key ideas explained in a chapter. One way to look at them 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.

1. What is software?
2. Why is software important?
3. Where is software important?
4. What could go wrong if some software fails? List some examples.
5. Where does software play an important role? List some examples.
6. What are some jobs related to software development? List some.
7. What's the difference between computer science and programming?
8. Where in the design, construction, and use of a ship is software used?
9. What is a server farm?

10. What kinds of queries do you ask online? List some.
11. What are some uses of software in science? List some.
12. What are some uses of software in medicine? List some.
13. What are some uses of software in entertainment? List some.
14. What general properties do we expect from good software?
15. What does a software developer look like?
16. What are the stages of software development?
17. Why can software development be difficult? List some reasons.
18. What are some uses of software that make your life easier?
19. What are some uses of software that make your life more difficult?

Terms

These terms present 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.

affordability	customer	programmer
analysis	design	programming
blackboard	feedback	software
CAD/CAM	GUI	stereotype
communication	ideals	testing
correctness	implementation	user

Exercises

1. Pick an activity you do most days (such as going to class, eating dinner, or watching television). Make a list of ways computers are directly or indirectly involved.
2. Pick a profession, preferably one that you have some interest in or some knowledge of. Make a list of activities done by people in that profession that involve computers.
3. Swap your list from exercise 2 with a friend who picked a different profession and improve his or her list. When you have both done that, compare your results. Remember: There is no perfect solution to an open-ended exercise; improvements are always possible.
4. From your own experience, describe an activity that would not have been possible without computers.
5. Make a list of programs (software applications) that you have directly used. List only examples where you obviously interact with a program (such as when selecting a new song on an MP3 player) and not cases

where there just might happen to be a computer involved (such as turning the steering wheel of your car).

6. Make a list of ten activities that people do that do not involve computers in any way, even indirectly. This may be harder than you think!
7. Identify five tasks for which computers are not used today, but for which you think they will be used at some time in the future. Write a few sentences to elaborate on each one that you choose.
8. Write an explanation (at least 100 words, but fewer than 500) of why you would like to be a computer programmer. If, on the other hand, you are convinced that you would not like to be a programmer, explain that. In either case, present well-thought-out, logical arguments.
9. Write an explanation (at least 100 words, but fewer than 500) of what role other than programmer you'd like to play in the computer industry (independently of whether "programmer" is your first choice).
10. Do you think computers will ever develop to be conscious, thinking beings, capable of competing with humans? Write a short paragraph (at least 100 words) supporting your position.
11. List some characteristics that most successful programmers share. Then list some characteristics that programmers are popularly assumed to have.
12. Identify at least five kinds of applications for computer programs mentioned in this chapter and pick the one that you find the most interesting and that you would most likely want to participate in someday. Write a short paragraph (at least 100 words) explaining why you chose the one you did.
13. How much memory would it take to store (a) this page of text, (b) this chapter, (c) all of Shakespeare's work? Assume one byte of memory holds one character and just try to be precise to about 20%.
14. How much memory does your computer have? Main memory? Disk?

Postscript

Our civilization runs on software. Software is an area of unsurpassed diversity and opportunities for interesting, socially useful, and profitable work. When you approach software, do it in a principled and serious manner: you want to be part of the solution, not add to the problems.

We are obviously in awe of the range of software that permeates our technological civilization. Not all applications of software do good, of course, but that is another story. Here we wanted to emphasize how pervasive software is and how much of what we rely on in our daily lives depends on software. It was all written by people like us. All the scientists, mathematicians, engineers, programmers, etc. who built the software briefly mentioned here started like you are starting.



Now, let's get back to the down-to-earth business of learning the technical skills needed to program. If you start wondering if it is worth all your hard work (most thoughtful people wonder about that sometime), come back and reread this chapter, the Preface, and bits of Chapter 0 (“Notes to the Reader”). If you start wondering if you can handle it all, remember that millions have succeeded in becoming competent programmers, designers, software engineers, etc. You can, too.

Part I

The Basics



Hello, World!

“Programming is learned
by writing programs.”

—Brian Kernighan

Here, we present the simplest C++ program that actually does anything. The purpose of writing this program is to

- Let you try your programming environment
- Give you a first feel of how you can get a computer to do things for you

Thus, we present the notion of a program, the idea of translating a program from human-readable form to machine instructions using a compiler, and finally executing those machine instructions.

- 2.1 Programs**
- 2.2 The classic first program**
- 2.3 Compilation**
- 2.4 Linking**
- 2.5 Programming environments**

2.1 Programs

To get a computer to do something, you (or someone else) have to tell it exactly – in excruciating detail – what to do. Such a description of “what to do” is called a *program*, and *programming* is the activity of writing and testing such programs.

In a sense, we have all programmed before. After all, we have given descriptions of tasks to be done, such as “how to drive to the nearest cinema,” “how to find the upstairs bathroom,” and “how to heat a meal in the microwave.” The difference between such descriptions and programs is one of degree of precision: humans tend to compensate for poor instructions by using common sense, but computers don’t. For example, “turn right in the corridor, up the stairs, it’ll be on your left” is probably a fine description of how to get to the upstairs bathroom. However, when you look at those simple instructions, you’ll find the grammar sloppy and the instructions incomplete. A human easily compensates. For example, assume that you are sitting at the table and ask for directions to the bathroom. You don’t need to be told to get up from your chair to get to the corridor, somehow walk around (and not across or under) the table, not to step on the cat, etc. You’ll not have to be told not to bring your knife and fork or to remember to switch on the light so that you can see the stairs. Opening the door to the bathroom before entering is probably also something you don’t have to be told.

In contrast, computers are *really* dumb. They have to have everything described precisely and in detail. Consider again “turn right in the corridor, up the stairs, it’ll be on your left.” Where is the corridor? What’s a corridor? What is “turn right”? What stairs? How do I go up stairs? (One step at a time? Two steps? Slide up the banister?) What is on my left? When will it be on my left? To be able to describe “things” precisely for a computer, we need a precisely defined language with a specific grammar (English is far too loosely structured for that) and a well-defined vocabulary for the kinds of actions we want performed. Such a language is called a *programming language*, and C++ is a programming language designed for a wide selection of programming tasks.

If you want greater philosophical detail about computers, programs, and programming, (re)read Chapter 1. Here, let’s have a look at some code, starting with a very simple program and the tools and techniques you need to get it to run.

2.2 The classic first program

Here is a version of the classic first program. It writes “Hello, World!” to your screen:

```
// This program outputs the message "Hello, World!" to the monitor

#include "std_lib_facilities.h"

int main()      // C++ programs start by executing the function main
{
    cout << "Hello, World!\n";   // output "Hello, World!"
    return 0;
}
```

Think of this text as a set of instructions that we give to the computer to execute, much as we would give a recipe to a cook to follow, or as a list of assembly instructions for us to follow to get a new toy working. Let’s discuss what each line of this program does, starting with the line

```
cout << "Hello, World!\n";           // output "Hello, World!"
```

That’s the line that actually produces the output. It prints the characters **Hello, World!** followed by a newline; that is, after writing **Hello, World!**, the cursor will be placed at the start of the next line. A *cursor* is a little blinking character or line showing where you can type the next character.

In C++, string literals are delimited by double quotes (""); that is, "**Hello, World!**\n" is a string of characters. The \n is a “special character” indicating a newline. The name **cout** refers to a standard output stream. Characters “put into **cout**” using the output operator << will appear on the screen. The name **cout** is pronounced “see-out” and is an abbreviation of “character **output stream**.” You’ll find abbreviations rather common in programming. Naturally, an abbreviation can be a bit of a nuisance the first time you see it and have to remember it, but once you start using abbreviations repeatedly, they become second nature, and they are essential for keeping program text short and manageable.

The end of that line

```
// output "Hello, World!"
```

is a comment. Anything written after the token // (that’s the character /, called “slash,” twice) on a line is a comment. Comments are ignored by the compiler and written for the benefit of programmers who read the code. Here, we used the comment to tell you what the beginning of that line actually did.

Comments are written to describe what the program is intended to do and in general to provide information useful for humans that can't be directly expressed in code. The person most likely to benefit from the comments in your code is you – when you come back to that code next week, or next year, and have forgotten exactly why you wrote the code the way you did. So, document your programs well. In §7.6.4, we'll discuss what makes good comments.



A program is written for two audiences. Naturally, we write code for computers to execute. However, we spend long hours reading and modifying the code. Thus, programmers are another audience for programs. So, writing code is also a form of human-to-human communication. In fact, it makes sense to consider the human readers of our code our primary audience: if they (we) don't find the code reasonably easy to understand, the code is unlikely to ever become correct. So, please don't forget: code is for reading – do all you can to make it readable. Anyway, the comments are for the benefit of human readers only; the computer doesn't look at the text in comments.

The first line of the program is a typical comment; it simply tells the human reader what the program is supposed to do:

```
// This program outputs the message "Hello, World!" to the monitor
```

Such comments are useful because the code itself says what the program does, not what we meant it to do. Also, we can usually explain (roughly) what a program should do to a human much more concisely than we can express it (in detail) in code to a computer. Often such a comment is the first part of the program we write. If nothing else, it reminds us what we are trying to do.

The next line

```
#include "std_lib_facilities.h"
```

is an “**#include** directive.” It instructs the computer to make available (“to include”) facilities from a file called **std_lib_facilities.h**. We wrote that file to simplify use of the facilities available in all implementations of C++ (“the C++ standard library”). We will explain its contents as we go along. It is perfectly ordinary standard C++, but it contains details that we'd rather not bother you with for another dozen chapters. For this program, the importance of **std_lib_facilities.h** is that we make the standard C++ stream I/O facilities available. Here, we just use the standard output stream, **cout**, and its output operator, **<<**. A file included using **#include** usually has the suffix **.h** and is called a *header* or a *header file*. A header contains definitions of terms, such as **cout**, that we use in our program.

How does a computer know where to start executing a program? It looks for a function called **main** and starts executing the instructions it finds there. Here is the function **main** of our “Hello, World!” program:

```
int main()    // C++ programs start by executing the function main
{
    cout << "Hello, World!\n";    // output "Hello, World!"
    return 0;
}
```

Every C++ program must have a function called **main** to tell it where to start executing. A function is basically a named sequence of instructions for the computer to execute in the order in which they are written. A function has four parts:

- A *return type*, here **int** (meaning “integer”), which specifies what kind of result, if any, the function will return to whoever asked for it to be executed. The word **int** is a reserved word in C++ (a *keyword*), so **int** cannot be used as the name of anything else (see §A.3.1).
- A *name*, here **main**.
- A *parameter list* enclosed in parentheses (see §8.2 and §8.6), here **()**; in this case, the parameter list is empty.
- A *function body* enclosed in a set of “curly braces,” **{ }** , which lists the actions (called *statements*) that the function is to perform.

It follows that the minimal C++ program is simply

```
int main() {}
```

That’s not of much use, though, because it doesn’t do anything. The **main()** (“the main function”) of our “Hello, World!” program has two statements in its body:

```
cout << "Hello, World!\n";    // output "Hello, World!"
return 0;
```

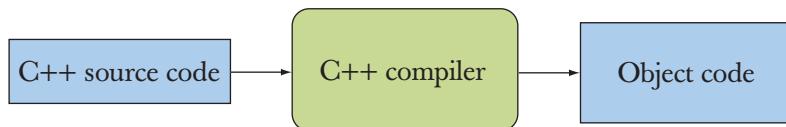
First it’ll write **Hello, World!** to the screen, and then it will return a value **0** (zero) to whoever called it. Since **main()** is called by “the system,” we won’t use that return value. However, on some systems (notably Unix/Linux) it can be used to check whether the program succeeded. A zero (**0**) returned by **main()** indicates that the program terminated successfully.

A part of a C++ program that specifies an action and isn’t an **#include** directive (or some other preprocessor directive; see §4.4 and §A.17) is called a *statement*.

2.3 Compilation

C++ is a compiled language. That means that to get a program to run, you must first translate it from the human-readable form to something a machine can

“understand.” That translation is done by a program called a *compiler*. What you read and write is called *source code* or *program text*, and what the computer executes is called *executable*, *object code*, or *machine code*. Typically C++ source code files are given the suffix **.cpp** (e.g., **hello_world.cpp**) or **.h** (as in **std_lib_facilities.h**), and object code files are given the suffix **.obj** (on Windows) or **.o** (Unix). The plain word *code* is therefore ambiguous and can cause confusion; use it with care only when it is obvious what’s meant by it. Unless otherwise specified, we use *code* to mean “source code” or even “the source code except the comments,” because comments really are there just for us humans and are not seen by the compiler generating object code.



The compiler reads your source code and tries to make sense of what you wrote. It looks to see if your program is grammatically correct, if every word has a defined meaning, and if there is anything obviously wrong that can be detected without trying to actually execute the program. You’ll find that compilers are rather picky about syntax. Leaving out any detail of our program, such as an **#include** file, a semicolon, or a curly brace, will cause errors. Similarly, the compiler has absolutely zero tolerance for spelling mistakes. Let us illustrate this with a series of examples, each of which has a single small error. Each error is an example of a kind of mistake we often make:

```
// no #include here
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

We didn’t include something to tell the compiler what **cout** was, so the compiler complains. To correct that, let’s add a header file:

```
#include "std_facilities.h"
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

Unfortunately, the compiler again complains: we misspelled `std_lib_facilities.h`. The compiler also objects to this:

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

We didn't terminate the string with a `"`. The compiler also objects to this:

```
#include "std_lib_facilities.h"
integer main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

The abbreviation `int` is used in C++ rather than the word `integer`. The compiler doesn't like this either:

```
#include "std_lib_facilities.h"
int main()
{
    cout < "Hello, World!\n";
    return 0;
}
```

We used `<` (the less-than operator) rather than `<<` (the output operator). The compiler also objects to this:

```
#include "std_lib_facilities.h"
int main()
{
    cout << 'Hello, World!\n';
    return 0;
}
```

We used single quotes rather than double quotes to delimit the string. Finally, the compiler gives an error for this:

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Hello, World!\n"
    return 0;
}
```

We forgot to terminate the output statement with a semicolon. Note that many C++ statements are terminated by a semicolon (`;`). The compiler needs those semicolons to know where one statement ends and the next begins. There is no really short, fully correct, and nontechnical way of summarizing where semicolons are needed. For now, just copy our pattern of use, which can be summarized as: “Put a semicolon after every expression that doesn’t end with a right curly brace (`)`.”

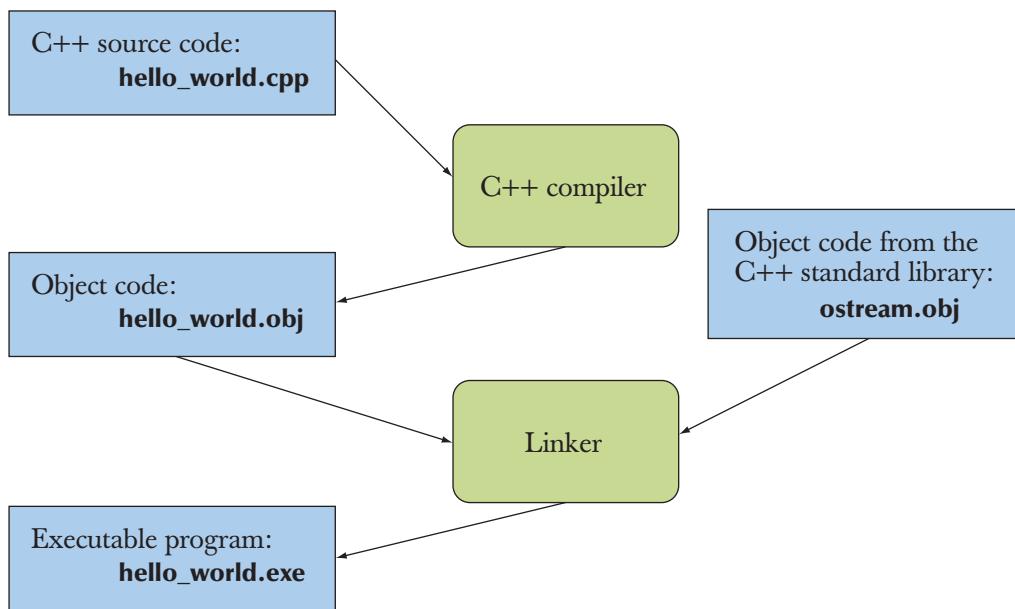
Why do we spend two pages of good space and minutes of your precious time showing you examples of trivial errors in a trivial program? To make the point that you – like all programmers – will spend a lot of time looking for errors in program source text. Most of the time, we look at text with errors in it. After all, if we were convinced that some code was correct, we’d typically be looking at some other code or taking the time off. It came as a major surprise to the early computer pioneers that they were making mistakes and had to devote a major portion of their time to finding them. It is still a surprise to most newcomers to programming.

When you program, you’ll get quite annoyed with the compiler at times. Sometimes it appears to complain about unimportant details (such as a missing semicolon) or about things you consider “obviously right.” However, the compiler is usually right: when it gives an error message and refuses to produce object code from your source code, there is something not quite right with your program; that is, the meaning of what you wrote isn’t precisely defined by the C++ standard.

The compiler has no common sense (it isn’t human) and is very picky about details. Since it has no common sense, you wouldn’t like it to try to guess what you meant by something that “looked OK” but didn’t conform to the definition of C++. If it did and its guess was different from yours, you could end up spending a lot of time trying to figure out why the program didn’t do what you thought you had told it to do. When all is said and done, the compiler saves us from a lot of self-inflicted problems. It saves us from many more problems than it causes. So, please remember: the compiler is your friend; possibly, the compiler is the best friend you have when you program.

2.4 Linking

A program usually consists of several separate parts, often developed by different people. For example, the “Hello, World!” program consists of the part we wrote plus parts of the C++ standard library. These separate parts (sometimes called *translation units*) must be compiled and the resulting object code files must be linked together to form an executable program. The program that links such parts together is (unsurprisingly) called a *linker*:



Please note that object code and executables are *not* portable among systems. For example, when you compile for a Windows machine, you get object code for Windows that will not run on a Linux machine.

A *library* is simply some code – usually written by others – that we access using declarations found in an **#included** file. A *declaration* is a program statement specifying how a piece of code can be used; we’ll examine declarations in detail later (e.g., §4.5.2).

Errors found by the compiler are called *compile-time errors*, errors found by the linker are called *link-time errors*, and errors not found until the program is run are called *run-time errors* or *logic errors*. Generally, compile-time errors are easier to understand and fix than link-time errors, and link-time errors are often easier to find and fix than run-time errors and logic errors. In Chapter 5 we discuss errors and the ways of handling them in greater detail.

2.5 Programming environments

To program, we use a programming language. We also use a compiler to translate our source code into object code and a linker to link our object code into an executable program. In addition, we use some program to enter our source code text into the computer and to edit it. These are just the first and most crucial tools that constitute our programmer’s tool set or “program development environment.”

If you work from a command-line window, as many professional programmers do, you will have to issue the compile and link commands yourself. If instead you use an IDE (“interactive development environment” or “integrated development environment”), as many professional programmers also do, a simple click on the correct button will do the job. See Appendix C for a description of how to compile and link on your C++ implementation.

IDEs usually include an editor with helpful features like color coding to help distinguish between comments, keywords, and other parts of your program source code, plus other facilities to help you debug your code, compile it, and run it. *Debugging* is the activity of finding errors in a program and removing them; you’ll hear a lot about that along the way.

Working with this book, you can use any system that provides an up-to-date, standards-conforming implementation of C++. Most of what we say will, with very minor modifications, be true for all implementations of C++, and the code will run everywhere. In our work, we use several different implementations.



Drill

So far we have talked about programming, code, and tools (such as compilers). Now you have to get a program to run. This is a crucial point in this book and in learning to program. This is where you start to develop practical skills and good programming habits. The exercises for this chapter are focused on getting you acquainted with your software development environment. Once you get the “Hello, World!” program to run, you will have passed the first major milestone as a programmer.

The purpose of a drill is to establish or reinforce your practical programming skills and give you experience with programming environment tools. Typically, a drill is a sequence of modifications to a single program, “growing” it from something completely trivial to something that might be a useful part of a real program. A traditional set of exercises is designed to test your initiative, cleverness, or inventiveness. In contrast, a drill requires little invention from you. Typically, sequencing is crucial, and each individual step should be easy (or even trivial).

Please don't try to be clever and skip steps; on average that will slow you down or even confuse you.

You might think you understand everything you read and everything your Mentor or instructor told you, but repetition and practice are necessary to develop programming skills. In this regard, programming is like athletics, music, dance, or any skill-based craft. Imagine people trying to compete in any of those fields without regular practice. You know how well they would perform. Constant practice – for professionals that means lifelong constant practice – is the only way to develop and maintain a high-level practical skill.

So, never skip the drills, no matter how tempted you are; they are essential to the learning process. Just start with the first step and proceed, testing each step as you go to make sure you are doing it right.

Don't be alarmed if you don't understand every detail of the syntax you are using, and don't be afraid to ask for help from instructors or friends. Keep going, do all of the drills and many of the exercises, and all will become clear in due time.

So, here is your first drill:

1. Go to Appendix C and follow the steps required to set up a project. Set up an empty console C++ project called hello_world.
2. Type in **hello_world.cpp**, exactly as specified below, save it in your practice directory (folder), and include it in your hello_world project.

```
#include "std_lib_facilities.h"
int main()      // C++ programs start by executing the function main
{
    cout << "Hello, World!\n"; // output "Hello, World!"
    keep_window_open();       // wait for a character to be entered
    return 0;
}
```

The call to **keep_window_open()** is needed on some Windows machines to prevent them from closing the window before you have a chance to read the output. This is a peculiarity/feature of Windows, not of C++. We defined **keep_window_open()** in **std_lib_facilities.h** to simplify writing simple text programs.

How do you find **std_lib_facilities.h**? If you are in a course, ask your instructor. If not, download it from our support site www.stroustrup.com/Programming. But what if you don't have an instructor and no access to the web? In that case (only), replace the **#include** directive with

```
#include<iostream>
#include<string>
#include<vector>
#include<algorithm>
```

```
#include<cmath>
using namespace std;
inline void keep_window_open() { char ch; cin>>ch; }
```

This uses the standard library directly, will keep you going until Chapter 5, and will be explained in detail later (§8.7).

3. Compile and run the “Hello, World!” program. Quite likely, something didn’t work quite right. It very rarely does in a first attempt to use a new programming language or a new programming environment. Find the problem and fix it! This is a point where asking for help from a more experienced person is sensible, but be sure to understand what you are shown so that you can do it all by yourself before proceeding further.
4. By now, you have probably encountered some errors and had to correct them. Now is the time to get a bit better acquainted with your compiler’s error-detection and error-reporting facilities! Try the six errors from §2.3 to see how your programming environment reacts. Think of at least five more errors you might have made typing in your program (e.g., forget **keep_window_open()**, leave the Caps Lock key on while typing a word, or type a comma instead of a semicolon) and try each to see what happens when you try to compile and run those versions.

Review



The basic idea of these review questions is to give you a chance to see if you have noticed and understood the key points of the chapter. You may have to refer back to the text to answer a question; that’s normal and expected. You may have to reread whole sections; that too is normal and expected. However, if you have to reread the whole chapter or have problems with every review question, you should consider whether your style of learning is effective. Are you reading too fast? Should you stop and do some of the **Try this** suggestions? Should you study with a friend so that you can discuss problems with the explanations in the text?

1. What is the purpose of the “Hello, World!” program?
2. Name the four parts of a function.
3. Name a function that must appear in every C++ program.
4. In the “Hello, World!” program, what is the purpose of the line **return 0;**?
5. What is the purpose of the compiler?
6. What is the purpose of the **#include** directive?
7. What does a **.h** suffix at the end of a file name signify in C++?
8. What does the linker do for your program?
9. What is the difference between a source file and an object file?
10. What is an IDE and what does it do for you?
11. If you understand everything in the textbook, why is it necessary to practice?

Most review questions have a clear answer in the chapter in which they appear. However, we do occasionally include questions to remind you of relevant information from other chapters and sometimes even relating to the world outside this book. We consider that fair; there is more to writing good software and thinking about the implications of doing so than fits into an individual chapter or book.

Terms

These terms present 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.

//	executable	main()
<<	function	object code
C++	header	output
comment	IDE	program
compiler	#include	source code
compile-time error	library	statement
cout	linker	

You might like to gradually develop a glossary written in your own words. You can do that by repeating exercise 5 below for each chapter.

Exercises

We list drills separately from exercises; always complete the chapter drill before attempting an exercise. Doing so will save you time.

1. Change the program to output the two lines

Hello, programming!
Here we go!

2. Expanding on what you have learned, write a program that lists the instructions for a computer to find the upstairs bathroom, discussed in §2.1. Can you think of any more steps that a person would assume, but that a computer would not? Add them to your list. This is a good start in “thinking like a computer.” Warning: For most people, “go to the bathroom” is a perfectly adequate instruction. For someone with no experience with houses or bathrooms (imagine a stone-age person, somehow transported into your dining room) the list of necessary instructions could be *very* long. Please don’t use more than a page. For the benefit of the reader, you may add a short description of the layout of the house you are imagining.
3. Write a description of how to get from the front door of your dorm room, apartment, house, whatever, to the door of your classroom (assuming you

are attending some school; if you are not, pick another target). Have a friend try to follow the instructions and annotate them with improvements as he or she goes along. To keep friends, it may be a good idea to “field test” those instructions before giving them to a friend.

4. Find a good cookbook. Read the instructions for baking blueberry muffins (if you are in a country where “blueberry muffins” is a strange, exotic dish, use a more familiar dish instead). Please note that with a bit of help and instruction, most of the people in the world can bake delicious blueberry muffins. It is not considered advanced or difficult fine cooking. However, for the author, few exercises in this book are as difficult as this one. It is amazing what you can do with a bit of practice.
 - Rewrite those instructions so that each individual action is in its own numbered paragraph. Be careful to list all ingredients and all kitchen utensils used at each step. Be careful about crucial details, such as the desired oven temperature, preheating the oven, the preparation of the muffin pan, the way to time the cooking, and the need to protect your hands when removing the muffins from the oven.
 - Consider those instructions from the point of view of a cooking novice (if you are not one, get help from a friend who does not know how to cook). Fill in the steps that the book’s author (almost certainly an experienced cook) left out for being obvious.
 - Build a glossary of terms used. (What’s a muffin pan? What does preheating do? What do you mean by “oven”?)
 - Now bake some muffins and enjoy your results.
5. Write a definition for each of the terms from “Terms.” First try to see if you can do it without looking at the chapter (not likely), then look through the chapter to find definitions. You might find the difference between your first attempt and the book’s version interesting. You might consult some suitable online glossary, such as www.stroustrup.com/glossary.html. By writing your own definition before looking it up, you reinforce the learning you achieved through your reading. If you have to reread a section to form a definition, that just helps you to understand. Feel free to use your own words for the definitions, and make the definitions as detailed as you think reasonable. Often, an example after the main definition will be helpful. You may like to store the definitions in a file so that you can add to them from the “Terms” sections of later chapters.

Postscript

What's so important about the "Hello, World!" program? Its purpose is to get us acquainted with the basic tools of programming. We tend to do an extremely simple example, such as "Hello, World!," whenever we approach a new tool. That way, we separate our learning into two parts: first we learn the basics of our tools with a trivial program, and later we learn about more complicated programs without being distracted by our tools. Learning the tools and the language simultaneously is far harder than doing first one and then the other. This approach to simplifying learning a complex task by breaking it into a series of small (and more manageable) steps is not limited to programming and computers. It is common and useful in most areas of life, especially in those that involve some practical skill.





Objects, Types, and Values

“Fortune favors the prepared mind.”

—Louis Pasteur

This chapter introduces the basics of storing and using data in a program. To do so, we first concentrate on reading in data from the keyboard. After establishing the fundamental notions of objects, types, values, and variables, we introduce several operators and give many examples of use of variables of types **char**, **int**, **double**, and **string**.

3.1 Input	3.6 Composite assignment operators
3.2 Variables	3.6.1 An example: find repeated words
3.3 Input and type	3.7 Names
3.4 Operations and operators	3.8 Types and objects
3.5 Assignment and initialization	3.9 Type safety
3.5.1 An example: detect repeated words	3.9.1 Safe conversions
	3.9.2 Unsafe conversions

3.1 Input

The “Hello, World!” program just writes to the screen. It produces output. It does not read anything; it does not get input from its user. That’s rather a bore. Real programs tend to produce results based on some input we give them, rather than just doing exactly the same thing each time we execute them.

To read something, we need somewhere to read into; that is, we need somewhere in the computer’s memory to place what we read. We call such a “place” an *object*. An *object* is a region of memory with a *type* that specifies what kind of information can be placed in it. A named object is called a *variable*. For example, character strings are put into **string** variables and integers are put into **int** variables. You can think of an object as a “box” into which you can put a value of the object’s type:

int:
age: 42

This would represent an object of type **int** named **age** containing the integer value **42**. Using a string variable, we can read a string from input and write it out again like this:

```
// read and write a first name
#include "std_lib_facilities.h"

int main()
{
    cout << "Please enter your first name (followed by 'enter'):\n";
    string first_name;           // first_name is a variable of type string
    cin >> first_name;          // read characters into first_name
    cout << "Hello, " << first_name << "!\n";
}
```

The `#include` and the `main()` are familiar from Chapter 2. Since the `#include` is needed for all our programs (up to Chapter 12), we'll leave it out of our presentation to avoid distraction. Similarly, we'll sometimes present code that will work only if it is placed in `main()` or some other function, like this:

```
cout << "Please enter your first name (followed by 'enter'):\n";
```

We assume that you can figure out how to put such code into a complete program for testing.

The first line of `main()` simply writes out a message encouraging the user to enter a first name. Such a message is typically called a *prompt* because it prompts the user to take an action. The next lines define a variable of type `string` called `first_name`, read input from the keyboard into that variable, and write out a greeting. Let's look at those three lines in turn:

```
string first_name; // first_name is a variable of type string
```

This sets aside an area of memory for holding a string of characters and gives it the name `first_name`:

string:
first_name:

A statement that introduces a new name into a program and sets aside memory for a variable is called a *definition*.

The next line reads characters from input (the keyboard) into that variable:

```
cin >> first_name; // read characters into first_name
```

The name `cin` refers to the standard input stream (pronounced “see-in,” for “character `in`put”) defined in the standard library. The second operand of the `>>` operator (“get from”) specifies where that input goes. So, if we type some first name, say `Nicholas`, followed by a newline, the string `"Nicholas"` becomes the value of `first_name`:

string:
first_name: Nicholas

The newline is necessary to get the machine’s attention. Until a newline is entered (the Enter key is hit), the computer simply collects characters. That “delay” gives you the chance to change your mind, erase some characters, and replace them



with others before hitting Enter. The newline will not be part of the string stored in memory.

Having gotten the input string into `first_name`, we can use it:

```
cout << "Hello, " << first_name << "\n";
```

This prints `Hello`, followed by `Nicholas` (the value of `first_name`) followed by `!` and a newline (`\n`) on the screen:

Hello, Nicholas!

If we had liked repetition and extra typing, we could have written three separate output statements instead:

```
cout << "Hello, ";
cout << first_name;
cout << "\n";
```

However, we are indifferent typists, and – more importantly – strongly dislike needless repetition (because repetition provides opportunity for errors), so we combined those three output operations into a single statement.

Note the way we use quotes around the characters in `"Hello, "` but not in `first_name`. We use quotes when we want a literal string. When we don't quote, we refer to the value of something with a name. Consider:

```
cout << "first_name" << " is " << first_name;
```

Here, `"first_name"` gives us the ten characters `first_name` and plain `first_name` gives us the value of the variable `first_name`, in this case, `Nicholas`. So, we get

first_name is Nicholas

3.2 Variables

Basically, we can do nothing of interest with a computer without storing data in memory, the way we did it with the input string in the example above. The “places” in which we store data are called *objects*. To access an object we need a *name*. A named object is called a *variable* and has a specific *type* (such as `int` or `string`) that determines what can be put into the object (e.g., `123` can go into an `int` and `"Hello, World!\n"` can go into a `string`) and which operations can be applied (e.g., we can multiply `ints` using the `*` operator and compare `strings` using the `<=` operator). The data items we put into variables are called *values*. A statement that

defines a variable is (unsurprisingly) called a *definition*, and a definition can (and usually should) provide an initial value. Consider:

```
string name = "Annemarie";
int number_of_steps = 39;
```

You can visualize these variables like this:

int:		string:
number_of_steps:	39	name: Annemarie

You cannot put values of the wrong type into a variable:

```
string name2 = 39;           // error: 39 isn't a string
int number_of_steps = "Annemarie"; // error: "Annemarie" is not an int
```

The compiler remembers the type of each variable and makes sure that you use it according to its type, as specified in its definition.

C++ provides a rather large number of types (see §A.8). However, you can write perfectly good programs using only five of those:

```
int number_of_steps = 39;      // int for integers
double flying_time = 3.5;     // double for floating-point numbers
char decimal_point = '.';     // char for individual characters
string name = "Annemarie";    // string for character strings
bool tap_on = true;          // bool for logical variables
```

The reason for the name **double** is historical: **double** is short for “double-precision floating point.” Floating point is the computer’s approximation to the mathematical concept of a real number.

Note that each of these types has its own characteristic style of literals:

39	// int: an integer
3.5	// double: a floating-point number
'.'	// char: an individual character enclosed in single quotes
"Annemarie"	// string: a sequence of characters delimited by double quotes
true	// bool: either true or false

That is, a sequence of digits (such as **1234**, **2**, or **976**) denotes an integer, a single character in single quotes (such as '**1**', '@', or '**x**') denotes a character, a sequence of digits with a decimal point (such as **1.234**, **0.12**, or **.98**) denotes a floating-point value, and a sequence of characters enclosed in double quotes (such as "**1234**", "**Howdy!**", or "**Annemarie**") denotes a string. For a detailed description of literals see §A.2.

3.3 Input and type



The input operation `>>` (“get from”) is sensitive to type; that is, it reads according to the type of variable you read into. For example:

```
// read name and age
int main()
{
    cout << "Please enter your first name and age\n";
    string first_name;      // string variable
    int age;                // integer variable
    cin >> first_name;     // read a string
    cin >> age;             // read an integer
    cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

So, if you type in **Carlos 22** the `>>` operator will read **Carlos** into `first_name`, **22** into `age`, and produce this output:

Hello, Carlos (age 22)

Why won’t it read (all of) **Carlos 22** into `first_name`? Because, by convention, reading of strings is terminated by what is called *whitespace*, that is, space, newline, and tab characters. Otherwise, whitespace by default is ignored by `>>`. For example, you can add as many spaces as you like before a number to be read; `>>` will just skip past them and read the number.

If you type in **22 Carlos**, you’ll see something that might be surprising until you think about it. The **22** will be read into `first_name` because, after all, **22** is a sequence of characters. On the other hand, **Carlos** isn’t an integer, so it will not be read. The output will be **22** followed by **(age** followed by some random number, such as **-96739** or **0**. Why? You didn’t give `age` an initial value and you didn’t succeed in reading a value into it. Therefore, you get some “garbage value” that happened to be in that part of memory when you started executing. In §10.6, we look at ways to handle “input format errors.” For now, let’s just initialize `age` so that we get a predictable value if the input fails:

```
// read name and age (2nd version)
int main()
{
    cout << "Please enter your first name and age\n";
```

```
string first_name = "???"; // string variable
                           // ("???" means "don't know the name")
int age = -1;           // integer variable (-1 means "don't know the age")
cin >> first_name >> age; // read a string followed by an integer
cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

Now the input **22 Carlos** will output

Hello, 22 (age -1)

Note that we can read several values in a single input statement, just as we can write several values in a single output statement. Note also that `<<` is sensitive to type, just as `>>` is, so we can output the **int** variable **age** as well as the **string** variable **first_name** and the string literals "**Hello,** " and "**(age** " and "**)\n**".

A **string** read using `>>` is (by default) terminated by whitespace; that is, it reads a single word. But sometimes, we want to read more than one word. There are of course many ways of doing this. For example, we can read a name consisting of two words like this:

```
int main()
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second; // read two strings
    cout << "Hello, " << first << " " << second << '\n';
}
```

We simply used `>>` twice, once for each name. When we want to write the names to output, we must insert a space between them.

TRY THIS



Get the “name and age” example to run. Then, modify it to write out the age in months: read the input in years and multiply (using the `*` operator) by 12. Read the age into a **double** to allow for children who can be very proud of being five and a half years old rather than just five.

3.4 Operations and operators

In addition to specifying what values can be stored in a variable, the type of a variable determines what operations we can apply to it and what they mean. For example:

```

int count;                                // >> reads an integer into count
cin >> count;
string name;                                // >> reads a string into name
cin >> name;

int c2 = count+2;                      // + adds integers
string s2 = name + " Jr. ";                // + appends characters

int c3 = count-2;                      // - subtracts integers
string s3 = name - " Jr. ";                // error: - isn't defined for strings

```

By “error” we mean that the compiler will reject a program trying to subtract strings. The compiler knows exactly which operations can be applied to each variable and can therefore prevent many mistakes. However, the compiler doesn’t know which operations make sense to you for which values, so it will happily accept legal operations that yield results that may look absurd to you. For example:

```
int age = -100;
```

It may be obvious to you that you can’t have a negative age (why not?) but nobody told the compiler, so it’ll produce code for that definition.

Here is a table of useful operators for some common and useful types:

	bool	char	int	double	string
assignment	=	=	=	=	=
addition			+	+	
concatenation					+
subtraction			-	-	
multiplication			*	*	
division			/	/	
remainder (modulo)			%		
increment by 1			++	++	
decrement by 1			--	--	
increment by n			+= n	+= n	

	bool	char	int	double	string
add to end					$+=$
decrement by n			$-= n$	$-= n$	
multiply and assign			$*=$	$*=$	
divide and assign			$/=$	$/=$	
remainder and assign			$\%=$		
read from s into x	$s >> x$				
write x to s	$s << x$				
equals	$==$	$==$	$==$	$==$	$==$
not equal	$!=$	$!=$	$!=$	$!=$	$!=$
greater than	$>$	$>$	$>$	$>$	$>$
greater than or equal	\geq	\geq	\geq	\geq	\geq
less than	$<$	$<$	$<$	$<$	$<$
less than or equal	\leq	\leq	\leq	\leq	\leq

A blank square indicates that an operation is not directly available for a type (though there may be indirect ways of using that operation; see §3.9.1). We'll explain these operations, and more, as we go along. The key points here are that there are a lot of useful operators and that their meaning tends to be the same for similar types.

Let's try an example involving floating-point numbers:

```
// simple program to exercise operators
int main()
{
    cout << "Please enter a floating-point value: ";
    double n;
    cin >> n;
    cout << "n == " << n
        << "\nn+1 == " << n+1
        << "\nthree times n == " << 3*n
        << "\ntwice n == " << n+n
        << "\nn squared == " << n*n
        << "\nhalf of n == " << n/2
        << "\nsquare root of n == " << sqrt(n)
        << "\n!"; // another name for newline ("end of line") in output
}
```

Obviously, the usual arithmetic operations have their usual notation and meaning as we know them from primary school. Naturally, not everything we might want

to do to a floating-point number, such as taking its square root, is available as an operator. Many operations are represented as named functions. In this case, we use `sqrt()` from the standard library to get the square root of `n`: `sqrt(n)`. The notation is familiar from math. We'll use functions along the way and discuss them in some detail in §4.5 and §8.5.

TRY THIS



Get this little program to run. Then, modify it to read an `int` rather than a `double`. Note that `sqrt()` is not defined for an `int` so assign `n` to a `double` and take `sqrt()` of that. Also, “exercise” some other operations. Note that for `ints` `/` is integer division and `%` is remainder (modulo), so that `5/2` is `2` (and not `2.5` or `3`) and `5%2` is `1`. The definitions of integer `*`, `/`, and `%` guarantee that for two positive `ints` `a` and `b` we have `a/b * b + a%b == a`.

Strings have fewer operators, but as we'll see in Chapter 23, they have plenty of named operations. However, the operators they do have can be used conventionally. For example:

```
// read first and second name
int main()
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second;           // read two strings
    string name = first + ' ' + second; // concatenate strings
    cout << "Hello, " << name << '\n';
}
```

For strings `+` means concatenation; that is, when `s1` and `s2` are strings, `s1+s2` is a string where the characters from `s1` are followed by the characters from `s2`. For example, if `s1` has the value `"Hello"` and `s2` the value `"World"`, then `s1+s2` will have the value `"HelloWorld"`. Comparison of `strings` is particularly useful:

```
// read and compare names
int main()
{
    cout << "Please enter two names\n";
    string first;
    string second;
```

```

cin >> first >> second;      // read two strings
if (first == second) cout << "that's the same name twice\n";
if (first < second)
    cout << first << " is alphabetically before " << second << '\n';
if (first > second)
    cout << first << " is alphabetically after " << second << '\n';
}

```

Here, we used an **if**-statement, which will be explained in detail in §4.4.1.1, to select actions based on conditions.

3.5 Assignment and initialization

In many ways, the most interesting operator is assignment, represented as **=**. It gives a variable a new value. For example:

int a = 3; // a starts out with the value 3

a:	3
----	---

a = 4; // a gets the value 4 ("becomes 4")

a:	4
----	---

int b = a; // b starts out with a copy of a's value (that is, 4)

a:	4
b:	4

b = a+5; // b gets the value a+5 (that is, 9)

a:	4
b:	9

a = a+7; // a gets the value a+7 (that is, 11)

a:	11
b:	9

That last assignment deserves notice. First of all it clearly shows that **=** does not mean equals – clearly, **a** doesn't equal **a+7**. It means assignment, that is, to place a new value in a variable. What is done for **a=a+7** is the following:

1. First, get the value of **a**; that's the integer 4.
2. Next, add 7 to that 4, yielding the integer 11.
3. Finally, put that 11 into **a**.

We can also illustrate assignments using strings:

```
string a = "alpha"; // a starts out with the value "alpha"
```

a: alpha

```
a = "beta"; // a gets the value "beta" (becomes "beta")
```

a: beta

```
string b = a; // b starts out with a copy of a's value (that is, "beta")
```

a: beta

b: beta

```
b = a + "gamma"; // b gets the value a + "gamma" (that is, "betagamma")
```

a: beta

b: betagamma

```
a = a + "delta"; // a gets the value a + "delta" (that is, "betadelta")
```

a: betadelta

b: betagamma



Above, we use “starts out with” and “gets” to distinguish two similar, but logically distinct, operations:

- Initialization (giving a variable its initial value)
- Assignment (giving a variable a new value)

These operations are so similar that C++ allows us to use the same notation (the `=`) for both:

```
int y = 8; // initialize y with 8
x = 9; // assign 9 to x
```

```
string t = "howdy!"; // initialize t with "howdy!"
s = "G'day"; // assign "G'day" to s
```

However, logically assignment and initialization are different. You can tell the two apart by the type specification (like `int` or `string`) that always starts an initialization; an assignment does not have that. In principle, an initialization always finds

the variable empty. On the other hand, an assignment (in principle) must clear out the old value from the variable before putting in the new value. You can think of the variable as a kind of small box and the value as a concrete thing, such as a coin, that you put into it. Before initialization, the box is empty, but after initialization it always holds a coin so that to put a new coin in, you (i.e., the assignment operator) first have to remove the old one (“destroy the old value”). Things are not quite this literal in the computer’s memory, but it’s not a bad way of thinking of what’s going on.

3.5.1 An example: detect repeated words

Assignment is needed when we want to put a new value into an object. When you think of it, it is obvious that assignment is most useful when you do things many times. We need an assignment when we want to do something again with a different value. Let’s have a look at a little program that detects adjacent repeated words in a sequence of words. Such code is part of most grammar checkers:

```
int main()
{
    string previous = " ";      // previous word; initialized to "not a word"
    string current;             // current word
    while (cin>>current) {     // read a stream of words
        if (previous == current) // check if the word is the same as last
            cout << "repeated word: " << current << '\n';
        previous = current;
    }
}
```

This program is not the most helpful since it doesn’t tell where the repeated word occurred in the text, but it’ll do for now. We will look at this program line by line starting with

```
string current; // current word
```

This is the string variable into which we immediately read the current (i.e., most recently read) word using

```
while (cin>>current)
```

This construct, called a **while**-statement, is interesting in its own right, and we’ll examine it further in §4.4.2.1. The **while** says that the statement after **(cin>>current)** is to be repeated as long as the input operation **cin>>current** succeeds, and **cin>>current** will succeed as long as there are characters to read on the



standard input. Remember that for a `string, >>` reads whitespace-separated words. You terminate this loop by giving the program an end-of-input character (usually referred to as *end of file*). On a Windows machine, that's Ctrl+Z (Control and Z pressed together) followed by an Enter (return). On a Unix or Linux machine that's Ctrl+D (Control and D pressed together).

So, what we do is to read a word into `current` and then compare it to the previous word (stored in `previous`). If they are the same, we say so:

```
if (previous == current)      // check if the word is the same as last
    cout << "repeated word: " << current << '\n';
```

Then we have to get ready to do this again for the next word. We do that by copying the `current` word into `previous`:

```
previous = current;
```

This handles all cases provided that we can get started. What should this code do for the first word where we have no previous word to compare? This problem is dealt with by the definition of `previous`:

```
string previous = " ";      // previous word; initialized to "not a word"
```

The `" "` contains only a single character (the space character, the one we get by hitting the space bar on our keyboard). The input operator `>>` skips whitespace, so we couldn't possibly read that from input. Therefore, the first time through the `while`-statement, the test

```
if (previous == current)
```

fails (as we want it to).

One way of understanding program flow is to “play computer,” that is, to follow the program line for line, doing what it specifies. Just draw boxes on a piece of paper and write their values into them. Change the values stored as specified by the program.

TRY THIS

Execute this program yourself using a piece of paper. Use the input **The cat cat jumped**. Even experienced programmers use this technique to visualize the actions of small sections of code that somehow don't seem completely obvious.

TRY THIS

Get the “repeated word detection program” to run. Test it with the sentence **She she laughed He He He because what he did did did not look very very good good**. How many repeated words were there? Why? What is the definition of *word* used here? What is the definition of *repeated word*? (For example, is **She she** a repetition?)

3.6 Composite assignment operators

Incrementing a variable (that is, adding 1 to it) is so common in programs that C++ provides a special syntax for it. For example:

++counter

means

counter = counter + 1

There are many other common ways of changing the value of a variable based on its current value. For example, we might like to add 7 to it, to subtract 9, or to multiply it by 2. Such operations are also supported directly by C++. For example:

```
a += 7;      // means a = a+7
b -= 9;      // means b = b-9
c *= 2;      // means c = c*2
```

In general, for any binary operator **oper**, **a oper= b** means **a = a oper b** (§A.5). For starters, that rule gives us operators **+=**, **-=**, ***=**, **/=**, and **%=**. This provides a pleasantly compact notation that directly reflects our ideas. For example, in many application domains ***=** and **/=** are referred to as “scaling.”

3.6.1 An example: find repeated words

Consider the example of detecting repeated adjacent words above. We could improve that by giving an idea of where the repeated word was in the sequence. A simple variation of that idea simply counts the words and outputs the count for the repeated word:

```
int main()
{
```

```
int number_of_words = 0;
string previous = " ";           // not a word
string current;
while (cin>>current) {
    ++number_of_words;          // increase word count
    if (previous == current)
        cout << "word number " << number_of_words
            << " repeated: " << current << "\n";
    previous = current;
}
```

We start our word counter at 0. Each time we see a word, we increment that counter:

`++number_of_words;`

That way, the first word becomes number 1, the next number 2, and so on. We could have accomplished the same by saying

`number_of_words += 1;`

or even

`number_of_words = number_of_words + 1;`

but **`++number_of_words`** is shorter and expresses the idea of incrementing directly.

Note how similar this program is to the one from §3.5.1. Obviously, we just took the program from §3.5.1 and modified it a bit to serve our new purpose. That's a very common technique: when we need to solve a problem, we look for a similar problem and use our solution for that with suitable modification. Don't start from scratch unless you really have to. Using a previous version of a program as a base for modification often saves a lot of time, and we benefit from much of the effort that went into the original program.



3.7 Names

We name our variables so that we can remember them and refer to them from other parts of a program. What can be a name in C++? In a C++ program, a name starts with a letter and contains only letters, digits, and underscores. For example:

```
x  
number_of_elements  
Fourier_transform  
z2  
Polygon
```

The following are not names:

```
2x           // a name must start with a letter  
time$to$market // $ is not a letter, digit, or underscore  
Start menu    // space is not a letter, digit, or underscore
```

When we say “not names,” we mean that a C++ compiler will not accept them as names.

If you read system code or machine-generated code, you might see names starting with underscores, such as `_foo`. Never write those yourself; such names are reserved for implementation and system entities. By avoiding leading underscores, you will never find your names clashing with some name that the implementation generated.

Names are case sensitive; that is, uppercase and lowercase letters are distinct, so `x` and `X` are different names. This little program has at least four errors:

```
#include "std_lib_facilities.h"  
  
int Main()  
{  
    STRING s = "Goodbye, cruel world! ";  
    cOut << s << '\n';  
}
```

It is usually not a good idea to define names that differ only in the case of a character, such as `one` and `One`; that will not confuse a compiler, but it can easily confuse a programmer.

TRY THIS



Compile the “Goodbye, cruel world!” program and examine the error messages. Did the compiler find all the errors? What did it suggest as the problems? Did the compiler get confused and diagnose more than four errors? Remove the errors one by one, starting with the lexically first, and see how the error messages change (and improve).



The C++ language reserves many (about 85) names as “keywords.” We list them in §A.3.1. You can’t use those to name your variables, types, functions, etc. For example:

```
int if = 7; // error: if is a keyword
```

You can use names of facilities in the standard library, such as `string`, but you shouldn’t. Reuse of such a common name will cause trouble if you should ever want to use the standard library:

```
int string = 7; // this will lead to trouble
```



When you choose names for your variables, functions, types, etc., choose meaningful names; that is, choose names that will help people understand your program. Even you will have problems understanding what your program is supposed to do if you have littered it with variables with “easy to type” names like `x1`, `x2`, `s3`, and `p7`. Abbreviations and acronyms can confuse people, so use them sparingly. These acronyms were obvious to us when we wrote them, but we expect you’ll have trouble with at least one:

```
mtbf  
TLA  
myw  
NBV
```

We expect that in a few months, we’ll also have trouble with at least one.

Short names, such as `x` and `i`, are meaningful when used conventionally; that is, `x` should be a local variable or a parameter (see §4.5 and §8.4) and `i` should be a loop index (see §4.4.2.3).

Don’t use overly long names; they are hard to type, make lines so long that they don’t fit on a screen, and are hard to read quickly. These are probably OK:

```
partial_sum  
element_count  
stable_partition
```

These are probably too long:

```
the_number_of_elements  
remaining_free_slots_in_symbol_table
```

Our “house style” is to use underscores to separate words in an identifier, such as `element_count`, rather than alternatives, such as `elementCount` and `Element-Count`. We never use names with all capital letters, such as `ALL_CAPITAL LETTERS`,

because that's conventionally reserved for macros (§27.8 and §A.17.2), which we avoid. We use an initial capital letter for types we define, such as **Square** and **Graph**. The C++ language and standard library don't use the initial-capital-letter style, so it's **int** rather than **Int** and **string** rather than **String**. Thus, our convention helps to minimize confusion between our types and the standard ones.

Avoid names that are easy to mistype, misread, or confuse. For example:

Name	names	nameS
foo	fo0	fl
f1	fl	fi

The characters **0** (zero), **o** (lowercase **O**), **O** (uppercase **o**), **1** (one), **I** (uppercase **i**), and **l** (lowercase **L**) are particularly prone to cause trouble.

3.8 Types and objects

The notion of type is central to C++ and most other programming languages. Let's take a closer and slightly more technical look at types, specifically at the types of the objects in which we store our data during computation. It'll save time in the long run, and it may save you some confusion.

- A *type* defines a set of possible values and a set of operations (for an object).
- An *object* is some memory that holds a value of a given type.
- A *value* is a set of bits in memory interpreted according to a type.
- A *variable* is a named object.
- A *declaration* is a statement that gives a name to an object.
- A *definition* is a declaration that sets aside memory for an object.

Informally, we think of an object as a box into which we can put values of a given type. An **int** box can hold integers, such as **7**, **42**, and **-399**. A **string** box can hold character string values, such as "**Interoperability**", "**tokens: !@#\$%^&***", and "**Old MacDonald had a farm**". Graphically, we can think of it like this:

<code>int a = 7;</code>	a:	7
<code>int b = 9;</code>	b:	9
<code>char c = 'a';</code>	c:	a
<code>double x = 1.2;</code>	x:	1.2
<code>string s1 = "Hello, World!";</code>	s1:	13 Hello, World!
<code>string s2 = "1.2";</code>	s2:	3 1.2


```
double y = x;      // the value of y is undefined
double z = 2.0+x;  // the meaning of + and the value of z are undefined
}
```

An implementation is even allowed to give a hardware error when the uninitialized **x** is used. Always initialize your variables! There are a few – very few – exceptions to this rule, such as a variable we immediately use as the target of an input operation, but always to initialize is a good habit that'll save you a lot of grief.

Complete type safety is the ideal and therefore the general rule for the language. Unfortunately, a C++ compiler cannot guarantee complete type safety, but we can avoid type safety violations through a combination of good coding practice and run-time checks. The ideal is never to use language features that the compiler cannot prove to be safe: static type safety. Unfortunately, that's too restrictive for most interesting uses of programming. The obvious fallback, that the compiler implicitly generates code that checks for type safety violations and catches all of them, is beyond C++. When we decide to do things that are (type) unsafe, we must do some checking ourselves. We'll point out such cases as we get to them.

The ideal of type safety is incredibly important when writing code. That's why we spend time on it this early in the book. Please note the pitfalls and avoid them.



3.9.1 Safe conversions

In §3.4, we saw that we couldn't directly add **chars** or compare a **double** to an **int**. However, C++ provides an indirect way to do both. When needed, a **char** is converted to an **int** and an **int** is converted to a **double**. For example:

```
char c = 'x';
int i1 = c;
int i2 = 'x';
```

Here both **i1** and **i2** get the value **120**, which is the integer value of the character '**x**' in the most popular 8-bit character set, ASCII. This is a simple and safe way of getting the numeric representation of a character. We call this **char-to-int** conversion safe because no information is lost; that is, we can copy the resulting **int** back into a **char** and get the original value:

```
char c2 = i1;
cout << c << ' << i1 << ' << c2 << '\n';
```

This will print

x 120 x

In this sense – that a value is always converted to an equal value or (for **doubles**) to the best approximation of an equal value – these conversions are safe:

bool to **char**

bool to **int**

bool to **double**

char to **int**

char to **double**

int to **double**

The most useful conversion is **int** to **double** because it allows us to mix **ints** and **doubles** in expressions:

```
double d1 = 2.3;
double d2 = d1+2;      // 2 is converted to 2.0 before adding
if (d1 < 0)            // 0 is converted to 0.0 before comparison
    cout << "d1 is negative";
```

For a really large **int**, we can (for some computers) suffer a loss of precision when converting to **double**. This is a rare problem.

3.9.2 Unsafe conversions

Safe conversions are usually a boon to the programmer and simplify writing code. Unfortunately, C++ also allows for (implicit) unsafe conversions. By unsafe, we mean that a value can be implicitly turned into a value of another type that does not equal the original value. For example:

```
int main()
{
    int a = 20000;
    char c = a;      // try to squeeze a large int into a small char
    int b = c;
    if (a != b)      // != means "not equal"
        cout << "oops!: " << a << "!=" << b << '\n';
    else
        cout << "Wow! We have large characters\n";
}
```

Such conversions are also called “narrowing” conversions, because they put a value into an object that may be too small (“narrow”) to hold it. Unfortunately,

few compilers warn about the unsafe initialization of the `char` with an `int`. The problem is that an `int` is typically much larger than a `char`, so that it can (and in this case does) hold an `int` value that cannot be represented as a `char`. Try it to see what value `b` gets on your machine (32 is a common result); better still, experiment:

```
int main()
{
    double d = 0;
    while (cin>>d) {           // repeat the statements below
                                // as long as we type in numbers
        int i = d;              // try to squeeze a double into an int
        char c = i;              // try to squeeze an int into a char
        int i2 = c;              // get the integer value of the character
        cout << "d==" << d          // the original double
        << " i==" << i          // converted to int
        << " i2==" << i2         // int value of char
        << " char(" << c << ")\\n"; // the char
    }
}
```

The `while`-statement that we use to allow many values to be tried will be explained in §4.4.2.1.

TRY THIS



Run this program with a variety of inputs. Try small values (e.g., `2` and `3`); try large values (larger than `127`, larger than `1000`); try negative values; try `56`; try `89`; try `128`; try non-integer values (e.g., `56.9` and `56.2`). In addition to showing how conversions from `double` to `int` and conversions from `int` to `char` are done on your machine, this program shows you what character (if any) your machine will print for a given integer value.

You'll find that many input values produce "unreasonable" results. Basically, we are trying to put a gallon into a pint pot (about 4 liters into a 500ml glass). All of the conversions

- `double` to `int`
- `double` to `char`
- `double` to `bool`
- `int` to `char`

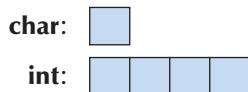
int to bool**char to bool**

are accepted by the compiler even though they are unsafe. They are unsafe in the sense that the value stored might differ from the value assigned. Why can this be a problem? Because often we don't suspect that an unsafe conversion is taking place. Consider:

```
double x = 2.7;
// lots of code
int y = x;           // y becomes 2
```

By the time we define **y** we may have forgotten that **x** was a **double**, or we may have temporarily forgotten that a **double**-to-**int** conversion truncates (always rounds down, toward zero) rather than using the conventional 4/5 rounding. What happens is perfectly predictable, but there is nothing in the **int y = x;** to remind us that information (the **.7**) is thrown away.

Conversions from **int** to **char** don't have problems with truncation – neither **int** nor **char** can represent a fraction of an integer. However, a **char** can hold only very small integer values. On a PC, a **char** is 1 byte whereas an **int** is 4 bytes:



So, we can't put a large number, such as 1000, into a **char** without loss of information: the value is “narrowed.” For example:

```
int a = 1000;
char b = a;           // b becomes -24 (on some machines)
```

Not all **int** values have **char** equivalents, and the exact range of **char** values depends on the particular implementation. On a PC the range of **char** values is [-128:127], but only [0:127] can be used portably because not every computer is a PC, and different computers have different ranges for their **char** values, such as [0:255].

Why do people accept the problem of narrowing conversions? The major reason is history: C++ inherited narrowing conversions from its ancestor language, C, so from day one of C++, there existed much code that depended on narrowing conversions. Also, many such conversions don't actually cause problems because the values involved happen to be in range, and many programmers object to compilers “telling them what to do.” In particular, the problems with unsafe conversions are often manageable in small programs and for experienced

programmers. They can be a source of errors in larger programs, though, and a significant cause of problems for novice programmers. However, compilers can warn about narrowing conversions – and many do.

C++11 introduced an initialization notation that outlaws narrowing conversions. For example, we could (and should) rewrite the troublesome examples above using a `{}`-list notation, rather than the `=` notation:

<code>double x {2.7};</code>	<code>// OK</code>
<code>int y {x};</code>	<code>// error: double -> int might narrow</code>
<code>int a {1000};</code>	<code>// OK</code>
<code>char b {a};</code>	<code>// error: int -> char might narrow</code>

When the initializer is an integer literal, the compiler can check the actual value and accept values that do not imply narrowing:

<code>int char b1 {1000};</code>	<code>// error: narrowing (assuming 8-bit chars)</code>
<code>char b2 {48};</code>	<code>// OK</code>

So what should you do if you think that a conversion might lead to a bad value? Use `{}` initializers to avoid accidents, and when you want a conversion, check the value before assigning as we did in the first example in this section. See §5.6.4 and §7.5 for a simplified way of doing such checking. The `{}`-list-based notation is known as *universal and uniform initialization* and we will see much more of that later on.



Drill

After each step of this drill, run your program to make sure it is really doing what you expect it to. Keep a list of what mistakes you make so that you can try to avoid those in the future.

1. This drill is to write a program that produces a simple form letter based on user input. Begin by typing the code from §3.1 prompting a user to enter his or her first name and writing “Hello, `first_name`” where `first_name` is the name entered by the user. Then modify your code as follows: change the prompt to “Enter the name of the person you want to write to” and change the output to “Dear `first_name`,”. Don’t forget the comma.
2. Add an introductory line or two, like “How are you? I am fine. I miss you.” Be sure to indent the first line. Add a few more lines of your choosing – it’s your letter.

3. Now prompt the user for the name of another friend, and store it in `friend_name`. Add a line to your letter: “Have you seen `friend_name` lately?”
4. Declare a `char` variable called `friend_sex` and initialize its value to 0. Prompt the user to enter an `m` if the friend is male and an `f` if the friend is female. Assign the value entered to the variable `friend_sex`. Then use two `if`-statements to write the following:

If the friend is male, write “If you see `friend_name` please ask him to call me.”

If the friend is female, write “If you see `friend_name` please ask her to call me.”

5. Prompt the user to enter the age of the recipient and assign it to an `int` variable `age`. Have your program write “I hear you just had a birthday and you are `age` years old.” If `age` is 0 or less or 110 or more, call `simple_error("you're kidding!")` using `simple_error()` from `std_lib_facilities.h`.
6. Add this to your letter:

If your friend is under 12, write “Next year you will be `age+1`.”

If your friend is 17, write “Next year you will be able to vote.”

If your friend is over 70, write “I hope you are enjoying retirement.”

Check your program to make sure it responds appropriately to each kind of value.

7. Add “Yours sincerely,” followed by two blank lines for a signature, followed by your name.

Review

1. What is meant by the term *prompt*?
2. Which operator do you use to read into a variable?
3. If you want the user to input an integer value into your program for a variable named `number`, what are two lines of code you could write to ask the user to do it and to input the value into your program?
4. What is `\n` called and what purpose does it serve?
5. What terminates input into a string?
6. What terminates input into an integer?
7. How would you write

```
cout << "Hello, ";
cout << first_name;
cout << "\n";
```

as a single line of code?

8. What is an object?
9. What is a literal?
10. What kinds of literals are there?
11. What is a variable?
12. What are typical sizes for a `char`, an `int`, and a `double`?
13. What measures do we use for the size of small entities in memory, such as `ints` and `strings`?
14. What is the difference between `=` and `==`?
15. What is a definition?
16. What is an initialization and how does it differ from an assignment?
17. What is string concatenation and how do you make it work in C++?
18. Which of the following are legal names in C++? If a name is not legal, why not?

`This_little_pig`
`latest thing`
`MiniMineMine`

`This_1_is_fine`
`the_$12_method`
`number`

`2_For_1_special`
`_this_is_ok`
`correct?`

19. Give five examples of legal names that you shouldn't use because they are likely to cause confusion.
20. What are some good rules for choosing names?
21. What is type safety and why is it important?
22. Why can conversion from `double` to `int` be a bad thing?
23. Define a rule to help decide if a conversion from one type to another is safe or unsafe.

Terms

assignment	definition	operation
<code>cin</code>	increment	operator
concatenation	initialization	type
conversion	name	type safety
declaration	narrowing	value
decrement	object	variable

Exercises

1. If you haven't done so already, do the **Try this** exercises from this chapter.
2. Write a program in C++ that converts from miles to kilometers. Your program should have a reasonable prompt for the user to enter a number of miles. Hint: There are 1.609 kilometers to the mile.
3. Write a program that doesn't do anything, but declares a number of variables with legal and illegal names (such as `int double = 0;`), so that you can see how the compiler reacts.

4. Write a program that prompts the user to enter two integer values. Store these values in **int** variables named **val1** and **val2**. Write your program to determine the smaller, larger, sum, difference, product, and ratio of these values and report them to the user.
5. Modify the program above to ask the user to enter floating-point values and store them in **double** variables. Compare the outputs of the two programs for some inputs of your choice. Are the results the same? Should they be? What's the difference?
6. Write a program that prompts the user to enter three integer values, and then outputs the values in numerical sequence separated by commas. So, if the user enters the values **10 4 6**, the output should be **4, 6, 10**. If two values are the same, they should just be ordered together. So, the input **4 5 4** should give **4, 4, 5**.
7. Do exercise 6, but with three string values. So, if the user enters the values **Steinbeck, Hemingway, Fitzgerald**, the output should be **Fitzgerald, Hemingway, Steinbeck**.
8. Write a program to test an integer value to determine if it is odd or even. As always, make sure your output is clear and complete. In other words, don't just output **yes** or **no**. Your output should stand alone, like **The value 4 is an even number**. Hint: See the remainder (modulo) operator in §3.4.
9. Write a program that converts spelled-out numbers such as “zero” and “two” into digits, such as 0 and 2. When the user inputs a number, the program should print out the corresponding digit. Do it for the values 0, 1, 2, 3, and 4 and write out **not a number I know** if the user enters something that doesn't correspond, such as **stupid computer!**.
10. Write a program that takes an operation followed by two operands and outputs the result. For example:

+ 100 3.14

*** 4 5**

Read the operation into a string called **operation** and use an **if**-statement to figure out which operation the user wants, for example, **if (operation=="+"**). Read the operands into variables of type **double**. Implement this for operations called +, -, *, /, plus, minus, mul, and div with their obvious meanings.

11. Write a program that prompts the user to enter some number of pennies (1-cent coins), nickels (5-cent coins), dimes (10-cent coins), quarters (25-cent coins), half dollars (50-cent coins), and one-dollar coins (100-cent coins). Query the user separately for the number of each size

coin, e.g., “How many pennies do you have?” Then your program should print out something like this:

You have 23 pennies.
You have 17 nickels.
You have 14 dimes.
You have 7 quarters.
You have 3 half dollars.
The value of all of your coins is 573 cents.

Make some improvements: if only one of a coin is reported, make the output grammatically correct, e.g., **14 dimes** and **1 dime** (not **1 dimes**). Also, report the sum in dollars and cents, i.e., **\$5.73** instead of **573 cents**.

Postscript

Please don’t underestimate the importance of the notion of type safety. Types are at the center of most notions of correct programs, and some of the most effective techniques for constructing programs rely on the design and use of types – as you’ll see in Chapters 6 and 9, Parts II, III, and IV.



Computation

“If it doesn’t have
to produce correct results,
I can make it arbitrarily fast.”

—Gerald M. Weinberg

This chapter presents the basics of computation. In particular, we discuss how to compute a value from a set of operands (*expression*), how to choose among alternative actions (*selection*), and how to repeat a computation for a series of values (*iteration*). We also show how a particular sub-computation can be named and specified separately (a *function*). Our primary concern is to express computations in ways that lead to correct and well-organized programs. To help you perform more realistic computations, we introduce the **vector** type to hold sequences of values.

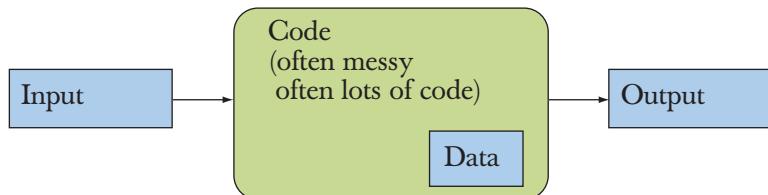
4.1 Computation
4.2 Objectives and tools
4.3 Expressions
4.3.1 Constant expressions
4.3.2 Operators
4.3.3 Conversions
4.4 Statements
4.4.1 Selection
4.4.2 Iteration

4.5 Functions
4.5.1 Why bother with functions?
4.5.2 Function declarations
4.6 vector
4.6.1 Traversing a vector
4.6.2 Growing a vector
4.6.3 A numeric example
4.6.4 A text example
4.7 Language features

4.1 Computation



From one point of view, all that a program ever does is to compute; that is, it takes some inputs and produces some output. After all, we call the hardware on which we run the program a computer. This view is accurate and reasonable as long as we take a broad view of what constitutes input and output:



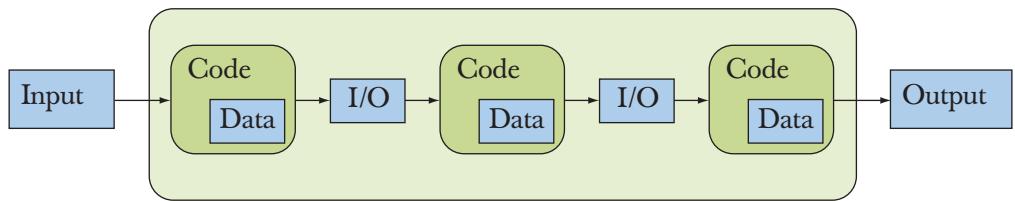
The input can come from a keyboard, from a mouse, from a touch screen, from files, from other input devices, from other programs, from other parts of a program. “Other input devices” is a category that contains most really interesting input sources: music keyboards, video recorders, network connections, temperature sensors, digital camera image sensors, etc. The variety is essentially infinite.

To deal with input, a program usually contains some data, sometimes referred to as its *data structures* or its *state*. For example, a calendar program may contain lists of holidays in various countries and a list of your appointments. Some of that data is part of the program from the start; other data is built up as the program reads input and collects useful information from it. For example, the calendar program will probably build your list of appointments from the input you give it. For the calendar, the main inputs are the requests to see the months and days you ask for (probably using mouse clicks) and the appointments you give it to keep track of (probably by typing information on your keyboard). The output is the display

of calendars and appointments, plus the buttons and prompts for input that the calendar program writes on your screen.

Input comes from a wide variety of sources. Similarly, output can go to a wide variety of destinations. Output can be to a screen, to files, to network connections, to other output devices, to other programs, and to other parts of a program. Examples of output devices include network interfaces, music synthesizers, electric motors, light generators, heaters, etc.

From a programming point of view the most important and interesting categories are “to/from another program” and “to/from other parts of a program.” Most of the rest of this book could be seen as discussing that last category: how do we express a program as a set of cooperating parts and how can they share and exchange data? These are key questions in programming. We can illustrate that graphically:



The abbreviation *I/O* stands for “input/output.” In this case, the output from one part of code is the input for the next part. What such “parts of a program” share is data stored in main memory, on persistent storage devices (such as disks), or transmitted over network connections. By “parts of a program” we mean entities such as a function producing a result from a set of input arguments (e.g., a square root from a floating-point number), a function performing an action on a physical object (e.g., a function drawing a line on a screen), or a function modifying some table within the program (e.g., a function adding a name to a table of customers).

When we say “input” and “output” we generally mean information coming into and out of a computer, but as you see, we can also use the terms for information given to or produced by a part of a program. Inputs to a part of a program are often called *arguments* and outputs from a part of a program are often called *results*.

By *computation* we simply mean the act of producing some outputs based on some inputs, such as producing the result (output) 49 from the argument (input) 7 using the computation (function) **square** (see §4.5). As a possibly helpful curiosity, we note that until the 1950s a computer was defined as a person who did computations, such as an accountant, a navigator, or a physicist. Today, we simply delegate most computations to computers (machines) of various forms, of which the pocket calculator is among the simplest.

4.2 Objectives and tools

Our job as programmers is to express computations

- Correctly
- Simply
- Efficiently

Please note the order of those ideals: it doesn't matter how fast a program is if it gives the wrong results. Similarly, a correct and efficient program can be so complicated that it must be thrown away or completely rewritten to produce a new version (release). Remember, useful programs will always be modified to accommodate new needs, new hardware, etc. Therefore a program – and any part of a program – should be as simple as possible to perform its task. For example, assume that you have written the perfect program for teaching basic arithmetic to children in your local school, and that its internal structure is a mess. Which language did you use to communicate with the children? English? English and Spanish? What if I'd like to use it in Finland? In Kuwait? How would you change the (natural) language used for communication with a child? If the internal structure of the program is a mess, the logically simple (but in practice almost always very difficult) operation of changing the natural language used to communicate with users becomes insurmountable.

Concerns about correctness, simplicity, and efficiency become ours the minute we start writing code for others and accept the responsibility to do that well; that is, we must accept that responsibility when we decide to become professionals. In practical terms, this means that we can't just throw code together until it appears to work; we must concern ourselves with the structure of code. Paradoxically, concerns for structure and "quality of code" are often the fastest ways of getting something to work. When programming is done well, such concerns minimize the need for the most frustrating part of programming: debugging; that is, good program structure during development can minimize the number of mistakes made and the time needed to search for such errors and to remove them.

Our main tool for organizing a program – and for organizing our thoughts as we program – is to break up a big computation into many little ones. This technique comes in two variations:

- *Abstraction:* Hide details that we don't need to use a facility ("implementation details") behind a convenient and general interface. For example, rather than considering the details of how to sort a phone book (thick books have been written about how to sort), we just call the `sort` algorithm from the C++ standard library. All we need to know to sort is how to invoke (call) that algorithm, so we can write `sort(b)` where `b` refers to

the phone book; **sort()** is a variant (§21.9) of the standard library **sort** algorithm (§21.8, §B.5.4) defined in **std_library.h**. Another example is the way we use computer memory. Direct use of memory can be quite messy, so we access it through typed and named variables (§3.2), standard library **vectors** (§4.6, Chapters 17–19), **maps** (Chapter 21), etc.

- “*Divide and conquer*”: Here we take a large problem and divide it into several little ones. For example, if we need to build a dictionary, we can separate that job into three: read the data, sort the data, and output the data. Each of the resulting problems is significantly smaller than the original.

Why does this help? After all, a program built out of parts is likely to be slightly larger than a program where everything is optimally merged together. The reason is that we are not very good at dealing with large problems. The way we actually deal with those – in programming and elsewhere – is to break them down into smaller problems, and we keep breaking those into even smaller parts until we get something simple enough to understand and solve. In terms of programming, you’ll find that a 1000-line program has far more than ten times as many errors as a 100-line program, so we try to compose the 1000-line program out of parts with fewer than 100 lines. For large programs, say 10,000,000 lines, applying abstraction and divide-and-conquer is not just an option, it’s an essential requirement. We simply cannot write and maintain large monolithic programs. One way of looking at the rest of this book is as a long series of examples of problems that need to be broken up into smaller parts together with the tools and techniques needed to do so.

When we consider dividing up a program, we must always consider what tools we have available to express the parts and their communications. A good library, supplying useful facilities for expressing ideas, can crucially affect the way we distribute functionality into different parts of a program. We cannot just sit back and “imagine” how best to partition a program; we must consider what libraries we have available to express the parts and their communication. It is early days yet, but not too soon to point out that if you can use an existing library, such as the C++ standard library, you can save yourself a lot of work, not just on programming but also on testing and documentation. The **iostreams** save us from having to directly deal with the hardware’s input/output ports. This is a first example of partitioning a program using abstraction. Every new chapter will provide more examples.

Note the emphasis on structure and organization: you don’t get good code just by writing a lot of statements. Why do we mention this now? At this stage you (or at least many readers) have little idea about what code is, and it will be months before you are ready to write code upon which other people could depend for their lives or livelihood. We mention it to help you get the emphasis of

your learning right. It is very tempting to dash ahead, focusing on the parts of programming that – like what is described in the rest of this chapter – are concrete and immediately useful and to ignore the “softer,” more conceptual parts of the art of software development. However, good programmers and system designers know (often having learned it the hard way) that concerns about structure lie at the heart of good software and that ignoring structure leads to expensive messes. Without structure, you are (metaphorically speaking) building with mud bricks. It can be done, but you’ll never get to the fifth floor (mud bricks lack the structural strength for that). If you have the ambition to build something reasonably permanent, you pay attention to matters of code structure and organization along the way, rather than having to come back and learn them after failures.

4.3 Expressions

The most basic building block of programs is an expression. An expression computes a value from a number of operands. The simplest expression is simply a literal value, such as **10**, **'a'**, **3.14**, or **"Norah"**.

Names of variables are also expressions. A variable represents the object of which it is the name. Consider:

```
// compute area:  
int length = 20;           // a literal integer (used to initialize a variable)  
int width = 40;  
int area = length*width;   // a multiplication
```

Here the literals **20** and **40** are used to initialize the variables **length** and **width**. Then, **length** and **width** are multiplied; that is, we multiply the values found in **length** and **width**. Here, **length** is simply shorthand for “the value found in the object named **length**.” Consider also

```
length = 99; // assign 99 to length
```

Here, as the left-hand operand of the assignment, **length** means “the object named **length**,” so that the assignment expression is read “Put **99** into the object named by **length**.” We distinguish between **length** used on the left-hand side of an assignment or an initialization (“the lvalue of **length**” or “the object named by **length**”) and **length** used on the right-hand side of an assignment or initialization (“the rvalue of **length**,” “the value of the object named by **length**,” or just “the value of **length**”). In this context, we find it useful to visualize a variable as a box labeled by its name:



That is, **length** is the name of an object of type **int** containing the value **99**. Sometimes (as an lvalue) **length** refers to the box (object) and sometimes (as an rvalue) **length** refers to the value in that box.

We can make more complicated expressions by combining expressions using operators, such as **+** and *****, in just the way that we are used to. When needed, we can use parentheses to group expressions:

```
int perimeter = (length+width)*2; // add then multiply
```

Without parentheses, we'd have had to say

```
int perimeter = length*2+width*2;
```

which is clumsy, and we might even have made this mistake:

```
int perimeter = length+width*2; // add width*2 to length
```

This last error is logical and cannot be found by the compiler. All the compiler sees is a variable called **perimeter** initialized by a valid expression. If the result of that expression is nonsense, that's your problem. You know the mathematical definition of a perimeter, but the compiler doesn't.

The usual mathematical rules of operator precedence apply, so **length+width*2** means **length+(width*2)**. Similarly **a*b+c/d** means **(a*b)+(c/d)** and not **a*(b+c)/d**. See §A.5 for a precedence table.

The first rule for the use of parentheses is simply “If in doubt, parenthesize,” but please do learn enough about expressions so that you are not in doubt about **a*b+c/d**. Overuse of parentheses, as in **(a*b)+(c/d)**, decreases readability.

Why should you care about readability? Because you and possibly others will read your code, and ugly code slows down reading and comprehension. Ugly code is not just hard to read, it is also much harder to get correct. Ugly code often hides logical errors. It is slower to read and makes it harder to convince yourself – and others – that the ugly code is correct. Don’t write absurdly complicated expressions such as

```
a*b+c/d*(e-f/g)/h+7 // too complicated
```

and always try to choose meaningful names.

4.3.1 Constant expressions

Programs typically use a lot of constants. For example, a geometry program might use **pi** and an inch-to-centimeter conversion program will use a conversion factor such as **2.54**. Obviously, we want to use meaningful names for those constants (as we did for **pi**; we didn’t say **3.14159**). Similarly, we don’t want to change those

constants accidentally. Consequently, C++ offers the notion of a symbolic constant, that is, a named object to which you can't give a new value after it has been initialized. For example:

```
constexpr double pi = 3.14159;
pi = 7;           // error: assignment to constant
double c = 2*pi*r; // OK: we just read pi; we don't try to change it
```

Such constants are useful for keeping code readable. You might have recognized **3.14159** as an approximation to **pi** if you saw it in some code, but would you have recognized **299792458**? Also, if someone asked you to change some code to use **pi** with the precision of 12 digits for your computation, you could search for **3.14** in your code, but if someone incautiously had used **22/7**, you probably wouldn't find it. It would be much better just to change the definition of **pi** to use the more appropriate value:

```
constexpr double pi = 3.14159265359;
```

Consequently, we prefer not to use literals (except very obvious ones, such as **0** and **1**) in most places in our code. Instead, we use constants with descriptive names. Non-obvious literals in code (outside definitions of symbolic constants) are derisively referred to as *magic constants*.

In some places, such as **case** labels (§4.4.1.3), C++ requires a *constant expression*, that is, an expression with an integer value composed exclusively of constants. For example:

```
constexpr int max = 17;      // a literal is a constant expression
int val = 19;

max+2                      // a constant expression (a const int plus a literal)
val+2                      // not a constant expression: it uses a variable
```

And by the way, **299792458** is one of the fundamental constants of the universe: the speed of light in vacuum measured in meters per second. If you didn't instantly recognize that, why would you expect not to be confused and slowed down by other literals embedded in code? Avoid magic constants!

A **constexpr** symbolic constant must be given a value that is known at compile time. For example:

```
constexpr int max = 100;

void use(int n)
{
```

```

constexpr int c1 = max+7; // OK: c1 is 107
constexpr int c2 = n+7;    // error: we don't know the value of c2
// ...
}

```

To handle cases where the value of a “variable” that is initialized with a value that is not known at compile time but never changes after initialization, C++ offers a second form of constant (a **const**):

```

constexpr int max = 100;

void use(int n)
{
    constexpr int c1 = max+7; // OK: c1 is 107
    const int c2 = n+7;      // OK, but don't try to change the value of c2
    // ...
    c2 = 7;              // error: c2 is a const
}

```

Such “**const** variables” are very common for two reasons:

- C++98 did not have **constexpr**, so people used **const**.
- “Variables” that are not constant expressions (their value is not known at compile time) but do not change values after initialization are in themselves widely useful.

4.3.2 Operators

We just used the simplest operators. However, you will soon need more as you want to express more complex operations. Most operators are conventional, so we’ll just explain them later as needed and you can look up details if and when you find a need. Here is a list of the most common operators:

Name	Comment
f(a)	function call
++lval	pre-increment
--lval	pre-decrement
!a	not
-a	unary minus
a*b	multiply
a/b	divide

(continues)

Name	Comment
a%b	modulo (remainder) only for integer types
a+b	add
a-b	subtract
out<<b	write b to out where out is an ostream
in>>b	read from in into b where in is an istream
a<b	less than result is bool
a<=b	less than or equal result is bool
a>b	greater than result is bool
a>=b	greater than or equal result is bool
a==b	equal not to be confused with = result is bool
a!=b	not equal result is bool
a && b	logical and result is bool
a b	logical or result is bool
lval = a	assignment not to be confused with ==
lval *= a	compound assignment lval = lval*a; also for /, %, +, -

We used **lval** (short for “value that can appear on the left-hand side of an assignment”) where the operator modifies an operand. You can find a complete list in §A.5.

For examples of the use of the logical operators **&&** (and), **||** (or), and **!** (not), see §5.5.1, §7.7, §7.8.2, and §10.4.

Note that **a<b<c** means **(a<b)<c** and that **a<b** evaluates to a Boolean value: **true** or **false**. So, **a<b<c** will be equivalent to either **true<c** or **false<c**. In particular, **a<b<c** does not mean “Is **b** between **a** and **c**? ” as many have naively (and not unreasonably) assumed. Thus, **a<b<c** is basically a useless expression. Don’t write such expressions with two comparison operations, and be very suspicious if you find such an expression in someone else’s code – it is most likely an error.

An increment can be expressed in at least three ways:

++a
a+=1
a=a+1

Which notation should we use? Why? We prefer the first version, **++a**, because it more directly expresses the idea of incrementing. It says what we want to do (increment **a**) rather than how to do it (add **1** to **a** and then write the result to **a**). In general, a way of saying something in a program is better than another if it more directly expresses an idea. The result is more concise and easier for a reader to understand. If we wrote **a=a+1**, a reader could easily wonder whether we really

meant to increment by 1. Maybe we just mistyped `a=b+1`, `a=a+2`, or even `a=a-1`; with `++a` there are far fewer opportunities for such doubts. Please note that this is a logical argument about readability and correctness, not an argument about efficiency. Contrary to popular belief, modern compilers tend to generate exactly the same code from `a=a+1` as for `++a` when `a` is one of the built-in types. Similarly, we prefer `a*=scale` over `a=a*scale`.

4.3.3 Conversions

We can “mix” different types in expressions. For example, `2.5/2` is a **double** divided by an **int**. What does this mean? Do we do integer division or floating-point division? Integer division throws away the remainder; for example, `5/2` is `2`. Floating-point division is different in that there is no remainder to throw away; for example, `5.0/2.0` is `2.5`. It follows that the most obvious answer to the question “Is `2.5/2` integer division or floating-point division?” is “Floating-point, of course; otherwise we’d lose information.” We would prefer the answer `1.25` rather than `1`, and `1.25` is what we get. The rule (for the types we have presented so far) is that if an operator has an operand of type **double**, we use floating-point arithmetic yielding a **double** result; otherwise, we use integer arithmetic yielding an **int** result. For example:

`5/2` is `2` (not `2.5`)

`2.5/2` means `2.5/double(2)`, that is, `1.25`

`'a'+1` means `int('a')+1`

The notations `type(value)` and `type{value}` mean “convert `value` to `type` as if you were initializing a variable of type `type` with the value `value`.” In other words, if necessary, the compiler converts (“promotes”) **int** operands to **doubles** or **char** operands to **ints**. The `type{value}` notation prevents narrowing (§3.9.2), but the `type(value)` notation does not. Once the result has been calculated, the compiler may have to convert it (again) to use it as an initializer or the right hand of an assignment. For example:

```
double d = 2.5;
int i = 2;

double d2 = d/i;      // d2 == 1.25
int i2 = d/i;        // i2 == 1
int i3 {d/i};        // error: double -> int conversion may narrow (§3.9.2)

d2 = d/i;            // d2 == 1.25
i2 = d/i;            // i2 == 1
```

Beware that it is easy to forget about integer division in an expression that also contains floating-point operands. Consider the usual formula for converting degrees Celsius to degrees Fahrenheit: $f = 9/5 * c + 32$. We might write

```
double dc;
cin >> dc;
double df = 9/5*dc+32;      // beware!
```

Unfortunately, but quite logically, this does not represent an accurate temperature scale conversion: the value of **9/5** is **1** rather than the **1.8** we might have hoped for. To get the code mathematically correct, either **9** or **5** (or both) will have to be changed into a **double**. For example:

```
double dc;
cin >> dc;
double df = 9.0/5*dc+32;    // better
```

4.4 Statements

An expression computes a value from a set of operands using operators like the ones mentioned in §4.3. What do we do when we want to produce several values? When we want to do something many times? When we want to choose among alternatives? When we want to get input or produce output? In C++, as in many languages, you use language constructs called *statements* to express those things.

So far, we have seen two kinds of statements: expression statements and declarations. An expression statement is simply an expression followed by a semicolon. For example:

```
a = b;
++b;
```

Those are two expression statements. Note that the assignment **=** is an operator so that **a=b** is an expression and we need the terminating semicolon to make **a=b;** a statement. Why do we need those semicolons? The reason is largely technical. Consider:

```
a = b ++ b;    // syntax error: missing semicolon
```

Without the semicolon, the compiler doesn't know whether we mean **a=b++;** **b;** or **a=b; ++b;**. This kind of problem is not restricted to computer languages; consider the exclamation “man eating tiger!” Who is eating whom? Punctuation exists to eliminate such problems, for example, “man-eating tiger!”

When statements follow each other, the computer executes them in the order in which they are written. For example:

```
int a = 7;  
cout << a << '\n';
```

Here the declaration, with its initialization, is executed before the output expression statement.

In general, we want a statement to have some effect. Statements without effect are typically useless. For example:

```
1+2; // do an addition, but don't use the sum  
a*b; // do a multiplication, but don't use the product
```

Such statements without effects are typically logical errors, and compilers often warn against them. Thus, expression statements are typically assignments, I/O statements, or function calls.

We will mention one more type of statement: the “empty statement.” Consider the code:

```
if (x == 5);  
{ y = 3; }
```

This looks like an error, and it almost certainly is. The ; in the first line is not supposed to be there. But, unfortunately, this is a legal construct in C++. It is called an *empty statement*, a statement doing nothing. An empty statement before a semicolon is rarely useful. In this case, it has the unfortunate consequence of allowing what is almost certainly an error to be acceptable to the compiler, so it will not alert you to the error and you will have that much more difficulty finding it.

What will happen if this code is run? The compiler will test **x** to see if it has the value **5**. If this condition is true, the following statement (the empty statement) will be executed, with no effect. Then the program continues to the next line, assigning the value **3** to **y** (which is what you wanted to have happen if **x** equals **5**). If, on the other hand, **x** does not have the value **5**, the compiler will not execute the empty statement (still no effect) and will continue as before to assign the value **3** to **y** (which is not what you wanted to have happen unless **x** equals **5**). In other words, the **if**-statement doesn’t matter; **y** is going to get the value **3** regardless. This is a common error for novice programmers, and it can be difficult to spot, so watch out for it.

The next section is devoted to statements used to alter the order of evaluation to allow us to express more interesting computations than those we get by just executing statements in the order in which they were written.

4.4.1 Selection

In programs, as in life, we often have to select among alternatives. In C++, that is done using either an **if**-statement or a **switch**-statement.

4.4.1.1 if-statements

The simplest form of selection is an **if**-statement, which selects between two alternatives. For example:

```
int main()
{
    int a = 0;
    int b = 0;
    cout << "Please enter two integers\n";
    cin >> a >> b;

    if (a<b)      // condition
        // 1st alternative (taken if condition is true):
        cout << "max(" << a << "," << b <<") is " << b << "\n";

    else
        // 2nd alternative (taken if condition is false):
        cout << "max(" << a << "," << b <<") is " << a << "\n";
}
```

An **if**-statement chooses between two alternatives. If its condition is true, the first statement is executed; otherwise, the second statement is. This notion is simple.

Most basic programming language features are. In fact, most basic facilities in a programming language are just new notation for things you learned in primary school – or even before that. For example, you were probably told in kindergarten that to cross the street at a traffic light, you had to wait for the light to turn green: “If the traffic light is green, go” and “If the traffic light is red, wait.” In C++ that becomes something like

```
if (traffic_light==green) go();
```

and

```
if (traffic_light==red) wait();
```

So, the basic notion is simple, but it is also easy to use **if**-statements in a too-simple-minded manner. Consider what’s wrong with this program (apart from leaving out the **#include** as usual):


```

cout << "Please enter a length followed by a unit (c or i):\n";
cin >> length >> unit;

if (unit == 'i')
    cout << length << "in == " << cm_per_inch * length << "cm\n";
else if (unit == 'c')
    cout << length << "cm == " << length / cm_per_inch << "in\n";
else
    cout << "Sorry, I don't know a unit called '" << unit << "'\n";
}

```

We first test for `unit=='i'` and then for `unit=='c'` and if it isn't (either) we say, "Sorry." It may look as if we used an "else-if-statement," but there is no such thing in C++. Instead, we combined two `if`-statements. The general form of an `if`-statement is

`if (expression) statement else statement`

That is, an `if`, followed by an *expression* in parentheses, followed by a *statement*, followed by an `else`, followed by a *statement*. What we did was to use an `if`-statement as the `else` part of an `if`-statement:

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

For our program that gives this structure:

```

if (unit == 'i')
    ...
    // 1st alternative
else if (unit == 'c')
    ...
    // 2nd alternative
else
    ...
    // 3rd alternative

```

In this way, we can write arbitrarily complex tests and associate a statement with each alternative. However, please remember that one of the ideals for code is simplicity, rather than complexity. You don't demonstrate your cleverness by writing the most complex program. Rather, you demonstrate competence by writing the simplest code that does the job.

TRY THIS

Use the example above as a model for a program that converts yen, euros, and pounds into dollars. If you like realism, you can find conversion rates on the web.

4.4.1.2 switch-statements

Actually, the comparison of `unit` to '`i`' and to '`c`' is an example of the most common form of selection: a selection based on comparison of a value against several constants. Such selection is so common that C++ provides a special statement for it: the `switch`-statement. We can rewrite our example as

```
int main()
{
    constexpr double cm_per_inch = 2.54;      // number of centimeters in
                                                // an inch
    double length = 1;                         // length in inches or
                                                // centimeters
    char unit = 'a';
    cout << "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;
    switch (unit) {
        case 'i':
            cout << length << "in == " << cm_per_inch*length << "cm\n";
            break;
        case 'c':
            cout << length << "cm == " << length/cm_per_inch << "in\n";
            break;
        default:
            cout << "Sorry, I don't know a unit called '" << unit << "'\n";
            break;
    }
}
```

The `switch`-statement syntax is archaic but still clearer than nested `if`-statements, especially when we compare against many constants. The value presented in parentheses after the `switch` is compared to a set of constants. Each constant is



presented as part of a `case` label. If the value equals the constant in a `case` label, the statement for that case is chosen. Each case is terminated by a `break`. If the value doesn't match any of the `case` labels, the statement identified by the `default` label is chosen. You don't have to provide a default, but it is a good idea to do so unless you are absolutely certain that you have listed every alternative. If you don't already know, programming will teach you that it's hard to be absolutely certain (and right) about anything.

4.4.1.3 Switch technicalities

Here are some technical details about `switch`-statements:

1. The value on which we switch must be of an integer, `char`, or enumeration (§9.5) type. In particular, you cannot switch on a `string`.
2. The values in the `case` labels must be constant expressions (§4.3.1). In particular, you cannot use a variable in a `case` label.
3. You cannot use the same value for two `case` labels.
4. You can use several `case` labels for a single case.
5. Don't forget to end each `case` with a `break`. Unfortunately, the compiler probably won't warn you if you forget.

For example:

```
int main()           // you can switch only on integers, etc.
{
    cout << "Do you like fish?\n";
    string s;
    cin >> s;
    switch (s) {      // error: the value must be of integer, char, or enum type
        case "no":
            // ...
            break;
        case "yes":
            // ...
            break;
    }
}
```

To select based on a `string` you have to use an `if`-statement or a `map` (Chapter 21).

A `switch`-statement generates optimized code for comparing against a set of constants. For larger sets of constants, this typically yields more efficient code

than a collection of **if**-statements. However, this means that the **case** label values must be constants and distinct. For example:

```
int main()          // case labels must be constants
{
    // define alternatives:
    int y = 'y';      // this is going to cause trouble
    constexpr char n = 'n';
    constexpr char m = '?';
    cout << "Do you like fish?\n";
    char a;
    cin >> a;
    switch (a) {
        case n:
            // ...
            break;
        case y:           // error: variable in case label
            // ...
            break;
        case m:
            // ...
            break;
        case 'n':         // error: duplicate case label (n's value is 'n')
            // ...
            break;
        default:
            // ...
            break;
    }
}
```

Often you want the same action for a set of values in a switch. It would be tedious to repeat the action so you can label a single action by a set of **case** labels. For example:

```
int main()          // you can label a statement with several case labels
{
    cout << "Please enter a digit\n";
    char a;
    cin >> a;
```

```

switch (a) {
    case '0': case '2': case '4': case '6': case '8':
        cout << "is even\n";
        break;
    case '1': case '3': case '5': case '7': case '9':
        cout << "is odd\n";
        break;
    default:
        cout << "is not a digit\n";
        break;
}

```



The most common error with **switch**-statements is to forget to terminate a **case** with a **break**. For example:

```

int main() // example of bad code (a break is missing)
{
    constexpr double cm_per_inch = 2.54; // number of centimeters in
                                         // an inch
    double length = 1;                  // length in inches or
                                         // centimeters
    char unit = 'a';
    cout << "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;

    switch (unit) {
        case 'i':
            cout << length << "in == " << cm_per_inch*length << "cm\n";
        case 'c':
            cout << length << "cm == " << length/cm_per_inch << "in\n";
    }
}

```

Unfortunately, the compiler will accept this, and when you have finished case '**i**' you'll just “drop through” into case '**c**', so that if you enter **2i** the program will output

```

2in == 5.08cm
2cm == 0.787402in

```

You have been warned!

TRY THIS



Rewrite your currency converter program from the previous **Try this** to use a **switch**-statement. Add conversions from yuan and kroner. Which version of the program is easier to write, understand, and modify? Why?

4.4.2 Iteration

We rarely do something only once. Therefore, programming languages provide convenient ways of doing something several times. This is called *repetition* or – especially when you do something to a series of elements of a data structure – *iteration*.

4.4.2.1 while-statements

As an example of iteration, consider the first program ever to run on a stored-program computer (the EDSAC). It was written and run by David Wheeler in the computer laboratory in Cambridge University, England, on May 6, 1949, to calculate and print a simple list of squares like this:

```

0    0
1    1
2    4
3    9
4    16
...
98   9604
99   9801

```

Each line is a number followed by a “tab” character (`\t`), followed by the square of the number. A C++ version looks like this:

```

// calculate and print a table of squares 0–99
int main()
{
    int i = 0;           // start from 0
    while (i<100) {
        cout << i << '\t' << square(i) << '\n';
        ++i;             // increment i (that is, i becomes i+1)
    }
}

```

The notation `square(i)` simply means the square of `i`. We'll later explain how it gets to mean that (§4.5).

No, this first modern program wasn't actually written in C++, but the logic was as is shown here:

- We start with 0.
- We see if we have reached 100, and if so we are finished.
- Otherwise, we print the number and its square, separated by a tab (`\t`), increase the number, and try again.

Clearly, to do this we need

- A way to repeat some statement (to *loop*)
- A variable to keep track of how many times we have been through the loop (a *loop variable* or a *control variable*), here the `int` called **i**
- An initializer for the loop variable, here **0**
- A termination criterion, here that we want to go through the loop 100 times
- Something to do each time around the loop (the *body* of the loop)

The language construct we used is called a **while**-statement. Just following its distinguishing keyword, **while**, it has a condition “on top” followed by its body:

```
while (i<100)           // the loop condition testing the loop variable i
{
    cout << i << '\t' << square(i) << '\n';
    ++i;                 // increment the loop variable i
}
```

The loop body is a block (delimited by curly braces) that writes out a row of the table and increments the loop variable, **i**. We start each pass through the loop by testing if **i<100**. If so, we are not yet finished and we can execute the loop body. If we have reached the end, that is, if **i** is **100**, we leave the **while**-statement and execute what comes next. In this program the end of the program is next, so we leave the program.

The loop variable for a **while**-statement must be defined and initialized outside (before) the **while**-statement. If we fail to define it, the compiler will give us an error. If we define it, but fail to initialize it, most compilers will warn us, saying something like “local variable **i** not set,” but would be willing to let us execute the program if we insisted. Don’t insist! Compilers are almost certainly right when they warn about uninitialized variables. Uninitialized variables are a common source of errors. In this case, we wrote

```
int i = 0;      // start from 0
```

so all is well.

Basically, writing a loop is simple. Getting it right for real-world problems can be tricky, though. In particular, it can be hard to express the condition correctly and to initialize all variables so that the loop starts correctly.

TRY THIS



The character 'b' is `char('a'+1)`, 'c' is `char('a'+2)`, etc. Use a loop to write out a table of characters with their corresponding integer values:

```
a  97
b  98
...
z  122
```

4.4.2.2 Blocks

Note how we grouped the two statements that the `while` had to execute:

```
while (i<100) {
    cout << i << '\t' << square(i) << '\n';
    ++i;           // increment i (that is, i becomes i+1)
}
```

A sequence of statements delimited by curly braces `{` and `}` is called a *block* or a *compound statement*. A block is a kind of statement. The empty block `{ }` is sometimes useful for expressing that nothing is to be done. For example:

```
if (a<=b) {      // do nothing
}
else {           // swap a and b
    int t = a;
    a = b;
    b = t;
}
```

4.4.2.3 for-statements

Iterating over a sequence of numbers is so common that C++, like most other programming languages, has a special syntax for it. A `for`-statement is like a `while`-statement except that the management of the control variable is concentrated



at the top where it is easy to see and understand. We could have written the “first program” like this:

```
// calculate and print a table of squares 0–99
int main()
{
    for (int i = 0; i<100; ++i)
        cout << i << '\t' << square(i) << '\n';
}
```

This means “Execute the body with **i** starting at **0** incrementing **i** after each execution of the body until we reach **100**.” A **for**-statement is always equivalent to some **while**-statement. In this case

```
for (int i = 0; i<100; ++i)
    cout << i << '\t' << square(i) << '\n';
```

means

```
{
    int i = 0;           // the for-statement initializer
    while (i<100) {    // the for-statement condition
        cout << i << '\t' << square(i) << '\n'; // the for-statement body
        ++i;             // the for-statement increment
    }
}
```

 Some novices prefer **while**-statements and some novices prefer **for**-statements. However, using a **for**-statement yields more easily understood and more maintainable code whenever a loop can be defined as a **for**-statement with a simple initializer, condition, and increment operation. Use a **while**-statement only when that’s not the case.

 Never modify the loop variable inside the body of a **for**-statement. That would violate every reader’s reasonable assumption about what a loop is doing. Consider:

```
int main()
{
    for (int i = 0; i<100; ++i) { // for i in the [0:100) range
        cout << i << '\t' << square(i) << '\n';
        ++i; // what's going on here? It smells like an error!
    }
}
```

Anyone looking at this loop would reasonably assume that the body would be executed 100 times. However, it isn't. The `++i` in the body ensures that `i` is incremented twice each time around the loop so that we get an output only for the 50 even values of `i`. If we saw such code, we would assume it to be an error, probably caused by a sloppy conversion from a `while`-statement. If you want to increment by 2, say so:

```
// calculate and print a table of squares of even numbers in the [0:100) range
int main()
{
    for (int i = 0; i<100; i+=2)
        cout << i << '\t' << square(i) << '\n';
}
```

Please note that the cleaner, more explicit version is shorter than the messy one. That's typical.



TRY THIS



Rewrite the character value example from the previous **Try this** to use a `for`-statement. Then modify your program to also write out a table of the integer values for uppercase letters and digits.

There is also a simpler “range-`for`-loop” for traversing collections of data, such as `vectors`; see §4.6.

4.5 Functions

In the program above, what was `square(i)`? It is a call of a function. In particular, it is a call of the function called `square` with the argument `i`. A *function* is a named sequence of statements. A function can return a result (also called a *return value*). The standard library provides a lot of useful functions, such as the square root function `sqrt()` that we used in §3.4. However, we write many functions ourselves. Here is a plausible definition of `square`:

```
int square(int x)    // return the square of x
{
    return x*x;
}
```

The first line of this definition tells us that this is a function (that's what the parentheses mean), that it is called **square**, that it takes an **int** argument (here, called **x**), and that it returns an **int** (the type of the result always comes first in a function declaration); that is, we can use it like this:

```
int main()
{
    cout << square(2) << '\n';      // print 4
    cout << square(10) << '\n';     // print 100
}
```

We don't have to use the result of a function call, but we do have to give a function exactly the arguments it requires. Consider:

```
square(2);           // probably a mistake: unused return value
int v1 = square();   // error: argument missing
int v2 = square();   // error: parentheses missing
int v3 = square(1,2); // error: too many arguments
int v4 = square("two"); // error: wrong type of argument – int expected
```

Many compilers warn against unused results, and all give errors as indicated. You might think that a computer should be smart enough to figure out that by the string "**two**" you really meant the integer **2**. However, a C++ compiler deliberately isn't that smart. It is the compiler's job to do exactly what you tell it to do after verifying that your code is well formed according to the definition of C++. If the compiler guessed about what you meant, it would occasionally guess wrong, and you – or the users of your program – would be quite annoyed. You'll find it hard enough to predict what your code will do without having the compiler "help you" by second-guessing you.

The *function body* is the block (§4.4.2.2) that actually does the work.

```
{
    return x*x;      // return the square of x
}
```

For **square**, the work is trivial: we produce the square of the argument and return that as our result. Saying that in C++ is easier than saying it in English. That's typical for simple ideas. After all, a programming language is designed to state such simple ideas simply and precisely.

The syntax of a *function definition* can be described like this:

type identifier (parameter-list) function-body

That is, a type (the return type), followed by an identifier (the name of the function), followed by a list of parameters in parentheses, followed by the body of the function (the statements to be executed). The list of arguments required by the function is called a *parameter list* and its elements are called *parameters* (or *formal arguments*). The list of parameters can be empty, and if we don't want to return a result we give **void** (meaning "nothing") as the return type. For example:

```
void write_sorry()    // take no argument; return no value
{
    cout << "Sorry\n";
}
```

The language-technical aspects of functions will be examined more closely in Chapter 8.

4.5.1 Why bother with functions?

We define a function when we want a separate computation with a name because doing so

- Makes the computation logically separate
- Makes the program text clearer (by naming the computation)
- Makes it possible to use the function in more than one place in our program
- Eases testing

We'll see many examples of each of those reasons as we go along, and we'll occasionally mention a reason. Note that real-world programs use thousands of functions, some even hundreds of thousands of functions. Obviously, we would never be able to write or understand such programs if their parts (e.g., computations) were not clearly separated and named. Also, you'll soon find that many functions are repeatedly useful and you'd soon tire of repeating equivalent code. For example, you might be happy writing **x*x** and **7*7** and **(x+7)*(x+7)**, etc. rather than **square(x)** and **square(7)** and **square(x+7)**, etc. However, that's only because **square** is a very simple computation. Consider square root (called **sqrt** in C++): you prefer to write **sqrt(x)** and **sqrt(7)** and **sqrt(x+7)**, etc. rather than repeating the (somewhat complicated and many lines long) code for computing square root. Even better: you don't have to even look at the computation of square root because knowing that **sqrt(x)** gives the square root of **x** is sufficient.

In §8.5 we will address many function technicalities, but for now, we'll just give another example.

If we had wanted to make the loop in `main()` really simple, we could have written

```
void print_square(int v)
{
    cout << v << '\t' << v*v << '\n';
}

int main()
{
    for (int i = 0; i<100; ++i) print_square(i);
}
```

Why didn't we use the version using `print_square()`? That version is not significantly simpler than the version using `square()`, and note that

- `print_square()` is a rather specialized function that we could not expect to be able to use later, whereas `square()` is an obvious candidate for other uses
- `square()` hardly requires documentation, whereas `print_square()` obviously needs explanation

The underlying reason for both is that `print_square()` performs two logically separate actions:

- It prints.
- It calculates a square.

Programs are usually easier to write and to understand if each function performs a single logical action. Basically, the `square()` version is the better design.

Finally, why did we use `square(i)` rather than simply `i*i` in the first version of the problem? Well, one of the purposes of functions is to simplify code by separating out complicated calculations as named functions, and for the 1949 version of the program there was no hardware that directly implemented “multiply.” Consequently, in the 1949 version of the program, `i*i` was actually a fairly complicated calculation, similar to what you'd do by hand using a piece of paper. Also, the writer of that original version, David Wheeler, was the inventor of the function (then called a subroutine) in modern computing, so it seemed appropriate to use it here.

TRY THIS



Implement `square()` without using the multiplication operator; that is, do the `x*x` by repeated addition (start a variable result at `0` and add `x` to it `x` times). Then run some version of “the first program” using that `square()`.

4.5.2 Function declarations

Did you notice that all the information needed to call a function was in the first line of its definition? For example:

```
int square(int x)
```

Given that, we know enough to say

```
int x = square(44);
```

We don't really need to look at the function body. In real programs, we most often don't want to look at a function body. Why would we want to look at the body of the standard library `sqrt()` function? We know it calculates the square root of its argument. Why would we want to see the body of our `square()` function? Of course we might just be curious. But almost all of the time, we are just interested in knowing how to call a function – seeing the definition would just be distracting. Fortunately, C++ provides a way of supplying that information separate from the complete function definition. It is called a *function declaration*:

```
int square(int);           // declaration of square
double sqrt(double);      // declaration of sqrt
```

Note the terminating semicolons. A semicolon is used in a function declaration instead of the body used in the corresponding function definition:

```
int square(int x)          // definition of square
{
    return x*x;
}
```

So, if you just want to use a function, you simply write – or more commonly `#include` – its declaration. The function definition can be elsewhere. We'll discuss where that "elsewhere" might be in §8.3 and §8.7. This distinction between declarations and definitions becomes essential in larger programs where we use declarations to keep most of the code out of sight to allow us to concentrate on a single part of a program at a time (§4.2).

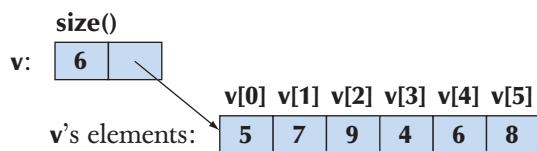
4.6 vector

To do just about anything of interest in a program, we need a collection of data to work on. For example, we might need a list of phone numbers, a list of members of a football team, a list of courses, a list of books read over the last year, a catalog

of songs for download, a set of payment options for a car, a list of the weather forecasts for the next week, a list of prices for a camera in different web stores, etc. The possibilities are literally endless and therefore ubiquitous in programs. We'll get to see a variety of ways of storing collections of data (a variety of containers of data; see Chapters 20 and 21). Here we will start with one of the simplest, and arguably the most useful, ways of storing data: a **vector**.



A **vector** is simply a sequence of elements that you can access by an index. For example, here is a **vector** called **v**:



That is, the first element has index 0, the second index 1, and so on. We refer to an element by subscripting the name of the **vector** with the element's index, so here the value of **v[0]** is 5, the value of **v[1]** is 7, and so on. Indices for a **vector** always start with 0 and increase by 1. This should look familiar: the standard library **vector** is simply the C++ standard library's version of an old and well-known idea. I have drawn the vector so as to emphasize that it "knows its size"; that is, a **vector** doesn't just store its elements, it also stores its size.

We could make such a **vector** like this:

```
vector<int> v = {5, 7, 9, 4, 6, 8}; // vector of 6 ints
```

We see that to make a **vector** we need to specify the type of the elements and the initial set of elements. The element type comes after **vector** in angle brackets (<**>**), here **<int>**. Here is another example:

```
vector<string> philosopher
= {"Kant", "Plato", "Hume", "Kierkegaard"}; // vector of 4 strings
```

Naturally, a **vector** will only accept elements of its declared element type:

```
philosopher[2] = 99; // error: trying to assign an int to a string
v[2] = "Hume"; // error: trying to assign a string to an int
```

We can also define a **vector** of a given size without specifying the element values. In that case, we use the (**n**) notation where **n** is the number of elements, and the elements are given a default value according to the element type. For example:

```
vector<int> vi(6); // vector of 6 ints initialized to 0
vector<string> vs(4); // vector of 4 strings initialized to ""
```

The string with no characters “” is called the empty string.

Please note that you cannot simply refer to a nonexistent element of a **vector**:

```
vi[20000] = 44;      // run-time error
```

We will discuss run-time errors and subscripting in the next chapter.

4.6.1 Traversing a vector

A **vector** “knows” its size, so we can print the elements of a **vector** like this:

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int i=0; i<v.size(); ++i)
    cout << v[i] << '\n';
```

The call **v.size()** gives the number of elements of the **vector** called **v**. In general, **v.size()** gives us the ability to access elements of a **vector** without accidentally referring to an element outside the **vector**’s range. The range for a **vector v** is **[0:v.size()]**. That’s the mathematical notation for a half-open sequence of elements. The first element of **v** is **v[0]** and the last **v[v.size()-1]**. If **v.size==0**, **v** has no elements, that is, **v** is an empty **vector**. This notion of half-open sequences is used throughout C++ and the C++ standard library (§17.3, §20.3).

The language takes advantage of the notion of a half-open sequence to provide a simple loop over all the elements of a sequence, such as the elements of a **vector**. For example:

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int x : v)      // for each x in v
    cout << x << '\n';
```

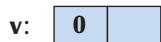
This is called a range-**for**-loop because the word *range* is often used to mean the same as “sequence of elements.” We read **for (int x : v)** as “for each **int x in v**” and the meaning of the loop is exactly like the equivalent loop over the subscripts **[0:v.size()]**. We use the range-**for**-loop for simple loops over all the elements of a sequence looking at one element at a time. More complicated loops, such as looking at every third element of a **vector**, looking at only the second half of a **vector**, or comparing elements of two **vectors**, are usually better done using the more complicated and more general traditional **for**-statement (§4.4.2.3).

4.6.2 Growing a vector

 Often, we start a **vector** empty and grow it to its desired size as we read or compute the data we want in it. The key operation here is **push_back()**, which adds a

new element to a **vector**. The new element becomes the last element of the **vector**. For example:

```
vector<double> v; // start off empty; that is, v has no elements
```



```
v.push_back(2.7); // add an element with the value 2.7 at end ("the back") of v
// v now has one element and v[0]==2.7
```



```
v.push_back(5.6); // add an element with the value 5.6 at end of v
// v now has two elements and v[1]==5.6
```



```
v.push_back(7.9); // add an element with the value 7.9 at end of v
// v now has three elements and v[2]==7.9
```



Note the syntax for a call of **push_back()**. It is called a *member function call*; **push_back()** is a member function of **vector** and must be called using this dot notation:

member-function-call:
object_name.member-function-name (argument-list)

The size of a **vector** can be obtained by a call to another of **vector**'s member functions: **size()**. Initially **v.size()** was **0**, and after the third call of **push_back()**, **v.size()** has become **3**.

If you have programmed before, you will note that a **vector** is similar to an array in C and other languages. However, you need not specify the size (length) of a **vector** in advance, and you can add as many elements as you like. As we go along, you'll find that the C++ standard **vector** has other useful properties.

4.6.3 A numeric example

Let's look at a more realistic example. Often, we have a series of values that we want to read into our program so that we can do something with them. The "something" could be producing a graph of the values, calculating the mean

and median, finding the largest element, sorting them, combining them with other data, searching for “interesting” values, comparing them to other data, etc. There is no limit to the range of computations we might perform on data, but first we need to get it into our computer’s memory. Here is the basic technique for getting an unknown – possibly large – amount of data into a computer. As a concrete example, we chose to read in floating-point numbers representing temperatures:

```
// read some temperatures into a vector
int main()
{
    vector<double> temps;           // temperatures
    for (double temp; cin>>temp; )   // read into temp
        temps.push_back(temp);       // put temp into vector
    // . . . do something . .
}
```

So, what goes on here? First we declare a **vector** to hold the data:

```
vector<double> temps;      // temperatures
```

This is where the type of input we expect is mentioned. We read and store **doubles**.

Next comes the actual read loop:

```
for (double temp; cin>>temp; )           // read into temp
    temps.push_back(temp);                 // put temp into vector
```

We define a variable **temp** of type **double** to read into. The **cin>>temp** reads a **double**, and that **double** is pushed into the **vector** (placed at the back). We have seen those individual operations before. What’s new here is that we use the input operation, **cin>>temp**, as the condition for a **for**-statement. Basically, **cin>>temp** is true if a value was read correctly and false otherwise, so that **for**-statement will read all the **doubles** we give it and stop when we give it anything else. For example, if you typed

```
1.2 3.4 5.6 7.8 9.0 |
```

then **temps** would get the five elements **1.2**, **3.4**, **5.6**, **7.8**, **9.0** (in that order, for example, **temps[0]==1.2**). We used the character ‘|’ to terminate the input – anything that isn’t a **double** can be used. In §10.6 we discuss how to terminate input and how to deal with errors in input.

To limit the scope of our input variable, **temp**, to the loop, we used a **for**-statement, rather than a **while**-statement:

```
double temp;
while (cin>>temp)           // read
    temps.push_back(temp);   // put into vector
// ... temp might be used here ...
```

As usual, a **for**-loop shows what is going on “up front” so that the code is easier to understand and accidental errors are harder to make.

Once we get data into a **vector** we can easily manipulate it. As an example, let’s calculate the mean and median temperatures:

```
// compute mean and median temperatures
int main()
{
    vector<double> temps;           // temperatures
    for (double temp; cin>>temp; ) // read into temp
        temps.push_back(temp);     // put temp into vector

    // compute mean temperature:
    double sum = 0;
    for (int x : temps) sum += x;
    cout << "Average temperature: " << sum/temps.size() << '\n';

    // compute median temperature:
    sort(temps);                  // sort temperatures
    cout << "Median temperature: " << temps[temps.size()/2] << '\n';
}
```

We calculate the average (the mean) by simply adding all the elements into **sum**, and then dividing the sum by the number of elements (that is, **temps.size()**):

```
// compute average temperature:
double sum = 0;
for (int x : temps) sum += x;
cout << "Average temperature: " << sum/temps.size() << '\n';
```

Note how the **`+=`** operator comes in handy.

To calculate a median (a value chosen so that half of the values are lower and the other half are higher) we need to sort the elements. For that, we use a variant of the standard library **sort** algorithm, **sort()**:

```
// compute median temperature:  
sort(temp);           // sort temperatures  
cout << "Median temperature: " << temps[temps.size()/2] << '\n';
```

We will explain the standard library algorithms much later (Chapter 20). Once the temperatures are sorted, it's easy to find the median: we just pick the middle element, the one with index `temp.size()/2`. If you feel like being picky (and if you do, you are starting to think like a programmer), you could observe that the value we found may not be a median according to the definition we offered above. Exercise 2 at the end of this chapter is designed to solve that little problem.

4.6.4 A text example

We didn't present the temperature example because we were particularly interested in temperatures. Many people – such as meteorologists, agronomists, and oceanographers – are very interested in temperature data and values based on it, such as means and medians. However, we are not. From a programmer's point of view, what's interesting about this example is its generality: the `vector` and the simple operations on it can be used in a huge range of applications. It is fair to say that whatever you are interested in, if you need to analyze data, you'll use `vector` (or a similar data structure; see Chapter 21). As an example, let's build a simple dictionary:

```
// simple dictionary: list of sorted words  
int main()  
{  
    vector<string> words;  
    for(string temp; cin>>temp; )      // read whitespace-separated words  
        words.push_back(temp);          // put into vector  
    cout << "Number of words: " << words.size() << '\n';  
  
    sort(words);                      // sort the words  
  
    for (int i = 0; i<words.size(); ++i)  
        if (i==0 || words[i-1]!=words[i]) // is this a new word?  
            cout << words[i] << "\n";  
}
```

If we feed some words to this program, it will write them out in order without repeating a word. For example, given

`a man a plan a canal panama`

it will write

```
a  
canal  
man  
panama  
plan
```

How do we stop reading string input? In other words, how do we terminate the input loop?

```
for (string temp; cin>>temp; )      // read  
    words.push_back(temp);           // put into vector
```

When we read numbers (in §4.6.2), we just gave some input character that wasn't a number. We can't do that here because every (ordinary) character can be read into a **string**. Fortunately, there are characters that are "not ordinary." As mentioned in §3.5.1, Ctrl+Z terminates an input stream under Windows and Ctrl+D does that under Unix.

Most of this program is remarkably similar to what we did for the temperatures. In fact, we wrote the "dictionary program" by cutting and pasting from the "temperature program." The only thing that's new is the test

```
if (i==0 || words[i-1]!=words[i])      // is this a new word?
```

If you deleted that test the output would be

```
a  
a  
a  
canal  
man  
panama  
plan
```

We didn't like the repetition, so we eliminated it using that test. What does the test do? It looks to see if the previous word we printed is different from the one we are about to print (**words[i-1]!=words[i]**) and if so, we print that word; otherwise, we do not. Obviously, we can't talk about a previous word when we are about

to print the first word (`i==0`), so we first test for that and combine those two tests using the `||` (or) operator:

```
if (i==0 || words[i-1]!=words[i]) // is this a new word?
```

Note that we can compare strings. We use `!=` (not equals) here; `==` (equals), `<` (less than), `<=` (less than or equal), `>` (greater than), and `>=` (greater than or equal) also work for strings. The `<`, `>`, etc. operators use the usual lexicographical ordering, so "Ape" comes before "Apple" and "Chimpanzee".

TRY THIS



Write a program that “bleeps” out words that you don’t like; that is, you read in words using `cin` and print them again on `cout`. If a word is among a few you have defined, you write out `BLEEP` instead of that word. Start with one “disliked word” such as

```
string disliked = "Broccoli";
```

When that works, add a few more.

4.7 Language features

The temperature and dictionary programs used most of the fundamental language features we presented in this chapter: iteration (the `for`-statement and the `while`-statement), selection (the `if`-statement), simple arithmetic (the `++` and `+=` operators), comparisons and logical operators (the `==`, `!=`, and `||` operators), variables, and functions (e.g., `main()`, `sort()`, and `size()`). In addition, we used standard library facilities, such as `vector` (a container of elements), `cout` (an output stream), and `sort()` (an algorithm).

If you count, you’ll find that we actually achieved quite a lot with rather few features. That’s the ideal! Each programming language feature exists to express a fundamental idea, and we can combine them in a huge (really, infinite) number of ways to write useful programs. This is a key notion: a computer is not a gadget with a fixed function. Instead it is a machine that we can program to do any computation we can think of, and given that we can attach computers to gadgets that interact with the world outside the computer, we can in principle get it to do anything.





Drill

Go through this drill step by step. Do not try to speed up by skipping steps. Test each step by entering at least three pairs of values – more values would be better.

1. Write a program that consists of a **while**-loop that (each time around the loop) reads in two **ints** and then prints them. Exit the program when a terminating '**|**' is entered.
2. Change the program to write out **the smaller value is:** followed by the smaller of the numbers and **the larger value is:** followed by the larger value.
3. Augment the program so that it writes the line **the numbers are equal** (only) if they are equal.
4. Change the program so that it uses **doubles** instead of **ints**.
5. Change the program so that it writes out **the numbers are almost equal** after writing out which is the larger and the smaller if the two numbers differ by less than 1.0/100.
6. Now change the body of the loop so that it reads just one **double** each time around. Define two variables to keep track of which is the smallest and which is the largest value you have seen so far. Each time through the loop write out the value entered. If it's the smallest so far, write **the smallest so far** after the number. If it is the largest so far, write **the largest so far** after the number.
7. Add a unit to each **double** entered; that is, enter values such as **10cm**, **2.5in**, **5ft**, or **3.33m**. Accept the four units: **cm**, **m**, **in**, **ft**. Assume conversion factors **1m == 100cm**, **1in == 2.54cm**, **1ft == 12in**. Read the unit indicator into a string. You may consider **12 m** (with a space between the number and the unit) equivalent to **12m** (without a space).
8. Reject values without units or with "illegal" representations of units, such as **y**, **yard**, **meter**, **km**, and **gallons**.
9. Keep track of the sum of values entered (as well as the smallest and the largest) and the number of values entered. When the loop ends, print the smallest, the largest, the number of values, and the sum of values. Note that to keep the sum, you have to decide on a unit to use for that sum; use meters.
10. Keep all the values entered (converted into meters) in a **vector**. At the end, write out those values.
11. Before writing out the values from the **vector**, sort them (that'll make them come out in increasing order).

Review

1. What is a computation?
2. What do we mean by inputs and outputs to a computation? Give examples.
3. What are the three requirements a programmer should keep in mind when expressing computations?
4. What does an expression do?
5. What is the difference between a statement and an expression, as described in this chapter?
6. What is an lvalue? List the operators that require an lvalue. Why do these operators, and not the others, require an lvalue?
7. What is a constant expression?
8. What is a literal?
9. What is a symbolic constant and why do we use them?
10. What is a magic constant? Give examples.
11. What are some operators that we can use for integers and floating-point values?
12. What operators can be used on integers but not on floating-point numbers?
13. What are some operators that can be used for **strings**?
14. When would a programmer prefer a **switch**-statement to an **if**-statement?
15. What are some common problems with **switch**-statements?
16. What is the function of each part of the header line in a **for**-loop, and in what sequence are they executed?
17. When should the **for**-loop be used and when should the **while**-loop be used?
18. How do you print the numeric value of a **char**?
19. Describe what the line **char foo(int x)** means in a function definition.
20. When should you define a separate function for part of a program? List reasons.
21. What can you do to an **int** that you cannot do to a **string**?
22. What can you do to a **string** that you cannot do to an **int**?
23. What is the index of the third element of a **vector**?
24. How do you write a **for**-loop that prints every element of a **vector**?
25. What does **vector<char> alphabet(26);** do?
26. Describe what **push_back()** does to a **vector**.
27. What do **vector**'s member functions **begin()**, **end()**, and **size()** do?
28. What makes **vector** so popular/useful?
29. How do you sort the elements of a **vector**?

Terms

abstraction	range-for-statement	push_back()
begin()	function	repetition
computation	if -statement	rvalue
conditional statement	increment	selection
declaration	input	size()
definition	iteration	sort()
divide and conquer	loop	statement
else	lvalue	switch -statement
end()	member function	vector
expression	output	while -statement
for -statement		

Exercises

1. If you haven't already, do the **Try this** exercises from this chapter.
2. If we define the median of a sequence as "a number so that exactly as many elements come before it in the sequence as come after it," fix the program in §4.6.3 so that it always prints out a median. Hint: A median need not be an element of the sequence.
3. Read a sequence of **double** values into a **vector**. Think of each value as the distance between two cities along a given route. Compute and print the total distance (the sum of all distances). Find and print the smallest and greatest distance between two neighboring cities. Find and print the mean distance between two neighboring cities.
4. Write a program to play a numbers guessing game. The user thinks of a number between 1 and 100 and your program asks questions to figure out what the number is (e.g., "Is the number you are thinking of less than 50?"). Your program should be able to identify the number after asking no more than seven questions. Hint: Use the **<** and **<=** operators and the **if-else** construct.
5. Write a program that performs as a very simple calculator. Your calculator should be able to handle the four basic math operations – add, subtract, multiply, and divide – on two input values. Your program should prompt the user to enter three arguments: two **double** values and a character to represent an operation. If the entry arguments are **35.6**, **24.1**, and **'+'**, the program output should be **The sum of 35.6 and 24.1 is 59.7**. In Chapter 6 we look at a much more sophisticated simple calculator.
6. Make a **vector** holding the ten **string** values **"zero"**, **"one"**, . . . **"nine"**. Use that in a program that converts a digit to its corresponding spelled-out value; e.g., the input **7** gives the output **seven**. Have the same

- program, using the same input loop, convert spelled-out numbers into their digit form; e.g., the input **seven** gives the output **7**.
7. Modify the “mini calculator” from exercise 5 to accept (just) single-digit numbers written as either digits or spelled out.
 8. There is an old story that the emperor wanted to thank the inventor of the game of chess and asked the inventor to name his reward. The inventor asked for one grain of rice for the first square, 2 for the second, 4 for the third, and so on, doubling for each of the 64 squares. That may sound modest, but there wasn’t that much rice in the empire! Write a program to calculate how many squares are required to give the inventor at least 1000 grains of rice, at least 1,000,000 grains, and at least 1,000,000,000 grains. You’ll need a loop, of course, and probably an **int** to keep track of which square you are at, an **int** to keep the number of grains on the current square, and an **int** to keep track of the grains on all previous squares. We suggest that you write out the value of all your variables for each iteration of the loop so that you can see what’s going on.
 9. Try to calculate the number of rice grains that the inventor asked for in exercise 8 above. You’ll find that the number is so large that it won’t fit in an **int** or a **double**. Observe what happens when the number gets too large to represent exactly as an **int** and as a **double**. What is the largest number of squares for which you can calculate the exact number of grains (using an **int**)? What is the largest number of squares for which you can calculate the approximate number of grains (using a **double**)?
 10. Write a program that plays the game “Rock, Paper, Scissors.” If you are not familiar with the game do some research (e.g., on the web using Google). Research is a common task for programmers. Use a **switch**-statement to solve this exercise. Also, the machine should give random answers (i.e., select the next rock, paper, or scissors randomly). Real randomness is too hard to provide just now, so just build a **vector** with a sequence of values to be used as “the next value.” If you build the **vector** into the program, it will always play the same game, so maybe you should let the user enter some values. Try variations to make it less easy for the user to guess which move the machine will make next.
 11. Create a program to find all the prime numbers between 1 and 100. One way to do this is to write a function that will check if a number is prime (i.e., see if the number can be divided by a prime number smaller than itself) using a **vector** of primes in order (so that if the **vector** is called **primes**, **primes[0]==2**, **primes[1]==3**, **primes[2]==5**, etc.). Then write a loop that goes from 1 to 100, checks each number to see if it is a prime, and stores each prime found in a **vector**. Write another loop that lists the primes you found. You might check your result by comparing your **vector** of prime numbers with **primes**. Consider 2 the first prime.

12. Modify the program described in the previous exercise to take an input value **max** and then find all prime numbers from **1** to **max**.
13. Create a program to find all the prime numbers between 1 and 100. There is a classic method for doing this, called the “Sieve of Eratosthenes.” If you don’t know that method, get on the web and look it up. Write your program using this method.
14. Modify the program described in the previous exercise to take an input value **max** and then find all prime numbers from **1** to **max**.
15. Write a program that takes an input value **n** and then finds the first **n** primes.
16. In the drill, you wrote a program that, given a series of numbers, found the max and min of that series. The number that appears the most times in a sequence is called the *mode*. Create a program that finds the mode of a set of positive integers.
17. Write a program that finds the min, max, and mode of a sequence of **strings**.
18. Write a program to solve quadratic equations. A quadratic equation is of the form

$$ax^2 + bx + c = 0$$

- If you don’t know the quadratic formula for solving such an expression, do some research. Remember, researching how to solve a problem is often necessary before a programmer can teach the computer how to solve it. Use **doubles** for the user inputs for **a**, **b**, and **c**. Since there are two solutions to a quadratic equation, output both **x1** and **x2**.
19. Write a program where you first enter a set of name-and-value pairs, such as **Joe 17** and **Barbara 22**. For each pair, add the name to a **vector** called **names** and the number to a **vector** called **scores** (in corresponding positions, so that if **names[7]=="Joe"** then **scores[7]==17**). Terminate input with **NoName 0**. Check that each name is unique and terminate with an error message if a name is entered twice. Write out all the (name,score) pairs, one per line.
 20. Modify the program from exercise 19 so that when you enter a name, the program will output the corresponding score or **name not found**.
 21. Modify the program from exercise 19 so that when you enter an integer, the program will output all the names with that score or **score not found**.

Postscript

From a philosophical point of view, you can now do everything that can be done using a computer – the rest is details! Among other things, this shows the value

of “details” and the importance of practical skills, because clearly you have barely started as a programmer. But we are serious. The tools presented in this chapter do allow you to express every computation: you have as many variables (including **vectors** and **strings**) as you want, you have arithmetic and comparisons, and you have selection and iteration. Every computation can be expressed using those primitives. You have text and numeric input and output, and every input or output can be expressed as text (even graphics). You can even organize your computations as sets of named functions. What is left for you to do is “just” to learn to write good programs, that is, to write programs that are correct, maintainable, and reasonably efficient. Importantly, you must try to learn to do so with a reasonable amount of effort.



Errors

“I realized that from now on a large part of my life would be spent finding and correcting my own mistakes.”

—Maurice Wilkes, 1949

In this chapter, we discuss correctness of programs, errors, and error handling. If you are a genuine novice, you’ll find the discussion a bit abstract at times and painfully detailed at other times. Can error handling really be this important? It is, and you’ll learn that one way or another before you can write programs that others are willing to use. What we are trying to do is to show you what “thinking like a programmer” is about. It combines fairly abstract strategy with painstaking analysis of details and alternatives.

5.1 Introduction	5.7 Logic errors
5.2 Sources of errors	5.8 Estimation
5.3 Compile-time errors	5.9 Debugging
5.3.1 Syntax errors	5.9.1 Practical debug advice
5.3.2 Type errors	
5.3.3 Non-errors	
5.4 Link-time errors	5.10 Pre- and post-conditions
5.5 Run-time errors	5.10.1 Post-conditions
5.5.1 The caller deals with errors	
5.5.2 The callee deals with errors	
5.5.3 Error reporting	
5.6 Exceptions	5.11 Testing
5.6.1 Bad arguments	
5.6.2 Range errors	
5.6.3 Bad input	
5.6.4 Narrowing errors	

5.1 Introduction

We have referred to errors repeatedly in the previous chapters, and – having done the drills and some exercises – you have some idea why. Errors are simply unavoidable when you develop a program, yet the final program must be free of errors, or at least free of errors that we consider unacceptable for it.

There are many ways of classifying errors. For example:

- *Compile-time errors*: Errors found by the compiler. We can further classify compile-time errors based on which language rules they violate, for example:
 - Syntax errors
 - Type errors
- *Link-time errors*: Errors found by the linker when it is trying to combine object files into an executable program.
- *Run-time errors*: Errors found by checks in a running program. We can further classify run-time errors as
 - Errors detected by the computer (hardware and/or operating system)
 - Errors detected by a library (e.g., the standard library)
 - Errors detected by user code
- *Logic errors*: Errors found by the programmer looking for the causes of erroneous results.

It is tempting to say that our job as programmers is to eliminate all errors. That is of course the ideal, but often that's not feasible. In fact, for real-world programs it can be hard to know exactly what "all errors" means. If we kicked out the power cord from your computer while it executed your program, would that be an error that you were supposed to handle? In many cases, the answer is "Obviously not," but what if we were talking about a medical monitoring program or the control program for a telephone switch? In those cases, a user could reasonably expect that something in the system of which your program was a part will do something sensible even if your computer lost power or a cosmic ray damaged the memory holding your program. The key question becomes: "Is my program supposed to detect that error?" Unless we specifically say otherwise, we will assume that your program

1. Should produce the desired results for all legal inputs
2. Should give reasonable error messages for all illegal inputs
3. Need not worry about misbehaving hardware
4. Need not worry about misbehaving system software
5. Is allowed to terminate after finding an error

Essentially all programs for which assumptions 3, 4, or 5 do not hold can be considered advanced and beyond the scope of this book. However, assumptions 1 and 2 are included in the definition of basic professionalism, and professionalism is one of our goals. Even if we don't meet that ideal 100% of the time, it must be the ideal.

When we write programs, errors are natural and unavoidable; the question is: How do we deal with them? Our guess is that avoiding, finding, and correcting errors takes 90% or more of the effort when developing serious software. For safety-critical programs, the effort can be greater still. You can do much better for small programs; on the other hand, you can easily do worse if you're sloppy.

Basically, we offer three approaches to producing acceptable software:

- Organize software to minimize errors.
- Eliminate most of the errors we made through debugging and testing.
- Make sure the remaining errors are not serious.

None of these approaches can completely eliminate errors by itself; we have to use all three.

Experience matters immensely when it comes to producing reliable programs, that is, programs that can be relied on to do what they are supposed to do with an acceptable error rate. Please don't forget that the ideal is that our programs always do the right thing. We are usually able only to approximate that ideal, but that's no excuse for not trying very hard.

5.2 Sources of errors

Here are some sources of errors:

- *Poor specification:* If we are not specific about what a program should do, we are unlikely to adequately examine the “dark corners” and make sure that all cases are handled (i.e., that every input gives a correct answer or an adequate error message).
- *Incomplete programs:* During development, there are obviously cases that we haven’t yet taken care of. That’s unavoidable. What we must aim for is to know when we have handled all cases.
- *Unexpected arguments:* Functions take arguments. If a function is given an argument we don’t handle, we have a problem. An example is calling the standard library square root function with -1.2 : `sqrt(-1.2)`. Since `sqrt()` of a `double` returns a `double`, there is no possible correct return value. §5.5.3 discusses this kind of problem.
- *Unexpected input:* Programs typically read data (from a keyboard, from files, from GUIs, from network connections, etc.). A program makes many assumptions about such input, for example, that the user will input a number. What if the user inputs “aw, shut up!” rather than the expected integer? §5.6.3 and §10.6 discuss this kind of problem.
- *Unexpected state:* Most programs keep a lot of data (“state”) around for use by different parts of the system. Examples are address lists, phone directories, and `vectors` of temperature readings. What if such data is incomplete or wrong? The various parts of the program must still manage. §26.3.5 discusses this kind of problem.
- *Logical errors:* That is, code that simply doesn’t do what it was supposed to do; we’ll just have to find and fix such problems. §6.6 and §6.9 give examples of finding such problems.

This list has a practical use. We can use it as a checklist when we are considering how far we have come with a program. No program is complete until we have considered all of these potential sources of errors. In fact, it is prudent to keep them in mind from the very start of a project, because it is most unlikely that a program that is just thrown together without thought about errors can have its errors found and removed without a serious rewrite.

5.3 Compile-time errors

When you are writing programs, your compiler is your first line of defense against errors. Before generating code, the compiler analyzes code to detect syntax errors and type errors. Only if it finds that the program completely conforms to the

language specification will it allow you to proceed. Many of the errors that the compiler finds are simply “silly errors” caused by mistyping or incomplete edits of the source code. Others result from flaws in our understanding of the way parts of our program interact. To a beginner, the compiler often seems petty, but as you learn to use the language facilities – and especially the type system – to directly express your ideas, you’ll come to appreciate the compiler’s ability to detect problems that would otherwise have caused you hours of tedious searching for bugs.

As an example, we will look at some calls of this simple function:

```
int area(int length, int width); // calculate area of a rectangle
```

5.3.1 Syntax errors

What if we were to call `area()` like this:

```
int s1 = area(7); // error: ) missing
int s2 = area(7) // error: ; missing
Int s3 = area(7); // error: Int is not a type
int s4 = area('7); // error: non-terminated character (' missing)
```

Each of those lines has a syntax error; that is, they are not well formed according to the C++ grammar, so the compiler will reject them. Unfortunately, syntax errors are not always easy to report in a way that you, the programmer, find easy to understand. That’s because the compiler may have to read a bit further than the error to be sure that there really is an error. The effect of this is that even though syntax errors tend to be completely trivial (you’ll often find it hard to believe you have made such a mistake once you find it), the reporting is often cryptic and occasionally refers to a line further on in the program. So, for syntax errors, if you don’t see anything wrong with the line the compiler points to, also look at previous lines in the program.

Note that the compiler has no idea what you are trying to do, so it cannot report errors in terms of your intent, only in terms of what you did. For example, given the error in the declaration of `s3` above, a compiler is unlikely to say

“You misspelled `int`; don’t capitalize the `i`.”

Rather, it’ll say something like

“syntax error: missing ‘;’ before identifier ‘`s3`’”
“`s3`’ missing storage-class or type identifiers”
“`Int`’ missing storage-class or type identifiers”

Such messages tend to be cryptic, until you get used to them, and to use a vocabulary that can be hard to penetrate. Different compilers can give very different-looking

error messages for the same code. Fortunately, you soon get used to reading such stuff. After all, a quick look at those cryptic lines can be read as

“There was a syntax error before **s3**,
and it had something to do with the type of **Int** or **s3**.”

Given that, it’s not rocket science to find the problem.

TRY THIS



Try to compile those examples and see how the compiler responds.

5.3.2 Type errors

Once you have removed syntax errors, the compiler will start reporting type errors; that is, it will report mismatches between the types you declared (or forgot to declare) for your variables, functions, etc. and the types of values or expressions you assign to them, pass as function arguments, etc. For example:

```
int x0 = arena(7);           // error: undeclared function
int x1 = area(7);            // error: wrong number of arguments
int x2 = area("seven",2);    // error: 1st argument has a wrong type
```

Let’s consider these errors.

1. For **arena(7)**, we misspelled **area** as **arena**, so the compiler thinks we want to call a function called **arena**. (What else could it “think”? That’s what we said.) Assuming there is no function called **arena()**, you’ll get an error message complaining about an undeclared function. If there is a function called **arena**, and if that function accepts **7** as an argument, you have a worse problem: the program will compile but do something you didn’t expect it to (that’s a logical error; see §5.7).
2. For **area(7)**, the compiler detects the wrong number of arguments. In C++, every function call must provide the expected number of arguments, of the right types, and in the right order. When the type system is used appropriately, this can be a powerful tool for avoiding run-time errors (see §14.1).
3. For **area("seven",2)**, you might hope that the computer would look at **"seven"** and figure out that you meant the integer **7**. It won’t. If a function needs an integer, you can’t give it a string. C++ does support some implicit type conversions (see §3.9) but not **string** to **int**. The compiler does not try to guess what you meant. What would you have expected for **area("Hovel lane",2)**, **area("7,2")**, and **area("sieben","zwei")**?

These are just a few examples. There are many more errors that the compiler will find for you.

TRY THIS



Try to compile those examples and see how the compiler responds. Try thinking of a few more errors yourself, and try those.

5.3.3 Non-errors

As you work with the compiler, you'll wish that it was smart enough to figure out what you meant; that is, you'd like some of the errors it reports not to be errors. That's natural. More surprisingly, as you gain experience, you'll begin to wish that the compiler would reject more code, rather than less. Consider:

```
int x4 = area(10,-7);      // OK: but what is a rectangle with a width of minus 7?  
int x5 = area(10.7,9.3);   // OK: but calls area(10,9)  
char x6 = area(100,9999); // OK: but truncates the result
```

For `x4` we get no error message from the compiler. From the compiler's point of view, `area(10,-7)` is fine: `area()` asks for two integers and you gave them to it; nobody said that those arguments had to be positive.

For `x5`, a good compiler will warn about the truncation of the `doubles` `10.7` and `9.3` into the `ints` `10` and `9` (see §3.9.2). However, the (ancient) language rules state that you can implicitly convert a `double` to an `int`, so the compiler is not allowed to reject the call `area(10.7,9.3)`.

The initialization of `x6` suffers from a variant of the same problem as the call `area(10.7,9.3)`. The `int` returned by `area(100,9999)`, probably `999900`, will be assigned to a `char`. The most likely result is for `x6` to get the “truncated” value `-36`. Again, a good compiler will give you a warning even though the (ancient) language rules prevent it from rejecting the code.

As you gain experience, you'll learn how to get the most out of the compiler's ability to detect errors and to dodge its known weaknesses. However, don't get overconfident: “my program compiled” doesn't mean that it will run. Even if it does run, it typically gives wrong results at first until you find the flaws in your logic.

5.4 Link-time errors

A program consists of several separately compiled parts, called *translation units*. Every function in a program must be declared with exactly the same type in



every translation unit in which it is used. We use header files to ensure that; see §8.3. Every function must also be defined exactly once in a program. If either of these rules is violated, the linker will give an error. We discuss how to avoid link-time errors in §8.3. For now, here is an example of a program that might give a typical linker error:

```
int area(int length, int width); // calculate area of a rectangle

int main()
{
    int x = area(2,3);
}
```

Unless we somehow have defined `area()` in another source file and linked the code generated from that source file to this code, the linker will complain that it didn't find a definition of `area()`.

The definition of `area()` must have exactly the same types (both the return type and the argument types) as we used in our file, that is:

```
int area(int x, int y) /* ... */ // "our" area()
```

Functions with the same name but different types will not match and will be ignored:

```
double area(double x, double y) /* ... */ // not "our" area()
```

```
int area(int x, int y, char unit) /* ... */ // not "our" area()
```

Note that a misspelled function name doesn't usually give a linker error. Instead, the compiler gives an error immediately when it sees a call to an undeclared function. That's good: compile-time errors are found earlier than link-time errors and are typically easier to fix.

The linkage rules for functions, as stated above, also hold for all other entities of a program, such as variables and types: there has to be exactly one definition of an entity with a given name, but there can be many declarations, and all have to agree exactly on its type. For more details, see §8.2–3.

5.5 Run-time errors

If your program has no compile-time errors and no link-time errors, it'll run. This is where the fun really starts. When you write the program you are able to detect

errors, but it is not always easy to know what to do with an error once you catch it at run time. Consider:

```
int area(int length, int width)      // calculate area of a rectangle
{
    return length*width;
}

int framed_area(int x, int y)        // calculate area within frame
{
    return area(x-2,y-2);
}

int main()
{
    int x = -1;
    int y = 2;
    int z = 4;
    // ...
    int area1 = area(x,y);
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = double(area1)/area3;    // convert to double to get
                                            // floating-point division
}
```

We used the variables `x`, `y`, `z` (rather than using the values directly as arguments) to make the problems less obvious to the human reader and harder for the compiler to detect. However, these calls lead to negative values, representing areas, being assigned to `area1` and `area2`. Should we accept such erroneous results, which violate most notions of math and physics? If not, who should detect the errors: the caller of `area()` or the function itself? And how should such errors be reported?

Before answering those questions, look at the calculation of the `ratio` in the code above. It looks innocent enough. Did you notice something wrong with it? If not, look again: `area3` will be `0`, so that `double(area1)/area3` divides by zero. This leads to a hardware-detected error that terminates the program with some cryptic message relating to hardware. This is the kind of error that you – or your users – will have to deal with if you don't detect and deal sensibly with run-time errors. Most people have low tolerance for such “hardware violations” because to anyone not intimately familiar with the program all the information provided is “Something went wrong somewhere!” That's insufficient for any constructive action, so we feel angry and would like to yell at whoever supplied the program.

So, let's tackle the problem of argument errors with `area()`. We have two obvious alternatives:

- a. Let the caller of `area()` deal with bad arguments.
- b. Let `area()` (the called function) deal with bad arguments.

5.5.1 The caller deals with errors

Let's try the first alternative ("Let the user beware!") first. That's the one we'd have to choose if `area()` was a function in a library where we couldn't modify it. For better or worse, this is the most common approach.

Protecting the call of `area(x,y)` in `main()` is relatively easy:

```
if (x<=0) error("non-positive x");
if (y<=0) error("non-positive y");
int area1 = area(x,y);
```

Really, the only question is what to do if we find an error. Here, we have called a function `error()` which we assume will do something sensible. In fact, in `std_lib_facilities.h` we supply an `error()` function that by default terminates the program with a system error message plus the string we passed as an argument to `error()`. If you prefer to write out your own error message or take other actions, you catch `runtime_error` (§5.6.2, §7.3, §7.8, §B.2.1). This approach suffices for most student programs and is an example of a style that can be used for more sophisticated error handling.

If we didn't need separate error messages about each argument, we would simplify:

```
if (x<=0 || y<=0) error("non-positive area() argument"); // || means "or"
int area1 = area(x,y);
```

To complete protecting `area()` from bad arguments, we have to deal with the calls through `framed_area()`. We could write

```
if (z<=2)
    error("non-positive 2nd area() argument called by framed_area()");
int area2 = framed_area(1,z);
if (y<=2 || z<=2)
    error("non-positive area() argument called by framed_area()");
int area3 = framed_area(y,z);
```

This is messy, but there is also something fundamentally wrong. We could write this only by knowing exactly how `framed_area()` used `area()`. We had to know that `framed_area()` subtracted 2 from each argument. We shouldn't have to know such details! What if someone modified `framed_area()` to use 1 instead of 2? Someone

doing that would have to look at every call of `framed_area()` and modify the error-checking code correspondingly. Such code is called “brittle” because it breaks easily. This is also an example of a “magic constant” (§4.3.1). We could make the code less brittle by giving the value subtracted by `framed_area()` a name:

```
constexpr int frame_width = 2;
int framed_area(int x, int y)    // calculate area within frame
{
    return area(x-frame_width,y-frame_width);
}
```

That name could be used by code calling `framed_area()`:

```
if (1-frame_width<=0 || z-frame_width<=0)
    error("non-positive argument for area() called by framed_area()");
int area2 = framed_area(1,z);
if (y-frame_width<=0 || z-frame_width<=0)
    error("non-positive argument for area() called by framed_area()");
int area3 = framed_area(y,z);
```

Look at that code! Are you sure it is correct? Do you find it pretty? Is it easy to read? Actually, we find it ugly (and therefore error-prone). We have more than trebled the size of the code and exposed an implementation detail of `framed_area()`. There has to be a better way!

Look at the original code:

```
int area2 = framed_area(1,z);
int area3 = framed_area(y,z);
```

It may be wrong, but at least we can see what it is supposed to do. We can keep this code if we put the check inside `framed_area()`.

5.5.2 The callee deals with errors

Checking for valid arguments within `framed_area()` is easy, and `error()` can still be used to report a problem:

```
int framed_area(int x, int y)    // calculate area within frame
{
    constexpr int frame_width = 2;
    if (x-frame_width<=0 || y-frame_width<=0)
        error("non-positive area() argument called by framed_area()");
    return area(x-frame_width,y-frame_width);
}
```

This is rather nice, and we no longer have to write a test for each call of `framed_area()`. For a useful function that we call 500 times in a large program, that can be a huge advantage. Furthermore, if anything to do with the error handling changes, we only have to modify the code in one place.

Note something interesting: we almost unconsciously slid from the “caller must check the arguments” approach to the “function must check its own arguments” approach (also called “the callee checks” because a called function is often called “a callee”). One benefit of the latter approach is that the argument-checking code is in one place. We don’t have to search the whole program for calls. Furthermore, that one place is exactly where the arguments are to be used, so all the information we need is easily available for us to do the check.

Let’s apply this solution to `area()`:

```
int area(int length, int width)      // calculate area of a rectangle
{
    if (length<=0 || width <=0) error("non-positive area() argument");
    return length*width;
}
```

This will catch all errors in calls to `area()`, so we no longer need to check in `framed_area()`. We might want to, though, to get a better – more specific – error message.

Checking arguments in the function seems so simple, so why don’t people do that always? Inattention to error handling is one answer, sloppiness is another, but there are also respectable reasons:

- *We can’t modify the function definition:* The function is in a library that for some reason can’t be changed. Maybe it’s used by others who don’t share your notions of what constitutes good error handling. Maybe it’s owned by someone else and you don’t have the source code. Maybe it’s in a library where new versions come regularly so that if you made a change, you’d have to change it again for each new release of the library.
- *The called function doesn’t know what to do in case of error:* This is typically the case for library functions. The library writer can detect the error, but only you know what is to be done when an error occurs.
- *The called function doesn’t know where it was called from:* When you get an error message, it tells you that something is wrong, but not how the executing program got to that point. Sometimes, you want an error message to be more specific.
- *Performance:* For a small function the cost of a check can be more than the cost of calculating the result. For example, that’s the case with `area()`, where the check also more than doubles the size of the function (that is,

the number of machine instructions that need to be executed, not just the length of the source code). For some programs, that can be critical, especially if the same information is checked repeatedly as functions call each other, passing information along more or less unchanged.

So what should you do? Check your arguments in a function unless you have a good reason not to.

After examining a few related topics, we'll return to the question of how to deal with bad arguments in §5.10.

5.5.3 Error reporting

Let's consider a slightly different question: Once you have checked a set of arguments and found an error, what should you do? Sometimes you can return an "error value." For example:

```
// ask user for a yes-or-no answer;
// return 'b' to indicate a bad answer (i.e., not yes or no)
char ask_user(string question)
{
    cout << question << "? (yes or no)\n";
    string answer = " ";
    cin >> answer;
    if (answer == "y" || answer == "yes") return 'y';
    if (answer == "n" || answer == "no") return 'n';
    return 'b'; // 'b' for "bad answer"
}

// calculate area of a rectangle;
// return -1 to indicate a bad argument
int area(int length, int width)
{
    if (length <= 0 || width <= 0) return -1;
    return length * width;
}
```

That way, we can have the called function do the detailed checking, while letting each caller handle the error as desired. This approach seems like it could work, but it has a couple of problems that make it unusable in many cases:

- Now both the called function and all callers must test. The caller has only a simple test to do but must still write that test and decide what to do if it fails.

- A caller can forget to test. That can lead to unpredictable behavior further along in the program.
- Many functions do not have an “extra” return value that they can use to indicate an error. For example, a function that reads an integer from input (such as `cin`'s operator `>>`) can obviously return any `int` value, so there is no `int` that it could return to indicate failure.

The second case above – a caller forgetting to test – can easily lead to surprises. For example:

```
int f(int x, int y, int z)
{
    int area1 = area(x,y);
    if (area1<=0) error("non-positive area");
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = double(area1)/area3;
    // ...
}
```

Do you see the errors? This kind of error is hard to find because there is no obvious “wrong code” to look at: the error is the absence of a test.

TRY THIS



Test this program with a variety of values. Print out the values of `area1`, `area2`, `area3`, and `ratio`. Insert more tests until all errors are caught. How do you know that you caught all errors? This is not a trick question; in this particular example you can give a valid argument for having caught all errors.

There is another solution that deals with that problem: using exceptions.

5.6 Exceptions

Like most modern programming languages, C++ provides a mechanism to help deal with errors: exceptions. The fundamental idea is to separate detection of an error (which should be done in a called function) from the handling of an error (which should be done in the calling function) while ensuring that a detected error cannot be ignored; that is, exceptions provide a mechanism that allows us to combine the best of the various approaches to error handling we have explored so far. Nothing makes error handling easy, but exceptions make it easier.



The basic idea is that if a function finds an error that it cannot handle, it does not **return** normally; instead, it **throws** an exception indicating what went wrong. Any direct or indirect caller can **catch** the exception, that is, specify what to do if the called code used **throw**. A function expresses interest in exceptions by using a **try**-block (as described in the following subsections) listing the kinds of exceptions it wants to handle in the **catch** parts of the **try**-block. If no caller catches an exception, the program terminates.

We'll come back to exceptions much later (Chapter 19) to see how to use them in slightly more advanced ways.

5.6.1 Bad arguments

Here is a version of **area()** using exceptions:

```
class Bad_area { };      // a type specifically for reporting errors from area()

// calculate area of a rectangle;
// throw a Bad_area exception in case of a bad argument
int area(int length, int width)
{
    if (length<=0 || width<=0) throw Bad_area();
    return length*width;
}
```

That is, if the arguments are OK, we return the area as always; if not, we get out of **area()** using the **throw**, hoping that some **catch** will provide an appropriate response. **Bad_area** is a new type we define with no other purpose than to provide something unique to **throw** from **area()** so that some **catch** can recognize it as the kind of exception thrown by **area()**. User-defined types (classes and enumeration) will be discussed in Chapter 9. The notation **Bad_area{}** means “Make an object of type **Bad_area** with the default value,” so **throw Bad_area{}** means “Make an object of type **Bad_area** and **throw** it.”

We can now write

```
int main()
try {
    int x = -1;
    int y = 2;
    int z = 4;
    ...
    int area1 = area(x,y);
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = area1/area3;
}
```

```
catch (Bad_area) {
    cout << "Oops! bad arguments to area()\n";
}
```

First note that this handles all calls to `area()`, both the one in `main()` and the two through `framed_area()`. Second, note how the handling of the error is cleanly separated from the detection of the error: `main()` knows nothing about which function did a `throw Bad_area{}`, and `area()` knows nothing about which function (if any) cares to `catch` the `Bad_area` exceptions it `throws`. This separation is especially important in large programs written using many libraries. In such programs, nobody can “just deal with an error by putting some code where it’s needed,” because nobody would want to modify code in both the application and in all of the libraries.

5.6.2 Range errors

Most real-world code deals with collections of data; that is, it uses all kinds of tables, lists, etc. of data elements to do a job. In the context of C++, we often refer to “collections of data” as *containers*. The most common and useful standard library container is the `vector` we introduced in §4.6. A `vector` holds a number of elements, and we can determine that number by calling the `vector`’s `size()` member function. What happens if we try to use an element with an index (subscript) that isn’t in the valid range `[0:v.size())`? The general notation `[low:high)` means indices from `low` to `high-1`, that is, including low but not high:



Before answering that question, we should pose another question and answer it:

“Why would you do that?” After all, you know that a subscript for `v` should be in the range `[0:v.size())`, so just be sure that’s so!

As it happens, that’s easy to say but sometimes hard to do. Consider this plausible program:

```
vector<int> v;                      // a vector of ints
for (int i; cin>>i; )
    v.push_back(i);                  // get values
for (int i = 0; i<=v.size(); ++i)    // print values
    cout << "v[" << i << "] == " << v[i] << '\n';
```

Do you see the error? Please try to spot it before reading on. It’s not an uncommon error. We have made such errors ourselves – especially late at night when

we were tired. Errors are always more common when you are tired or rushed. We use `0` and `size()` to try to make sure that `i` is always in range when we do `v[i]`.

Unfortunately, we made a mistake. Look at the `for`-loop: the termination condition is `i<=v.size()` rather than the correct `i<v.size()`. This has the unfortunate consequence that if we read in five integers we'll try to write out six. We try to read `v[5]`, which is one beyond the end of the `vector`. This kind of error is so common and “famous” that it has several names: it is an example of an *off-by-one error*, a *range error* because the index (subscript) wasn't in the range required by the `vector`, and a *bounds error* because the index was not within the limits (bounds) of the `vector`.

Why didn't we use a range-`for`-statement to express that loop? With a range-`for`, we cannot get the end of the loop wrong. However, for this loop, we wanted not only the value of each element but also the indices (subscripts). A range-`for` doesn't give that without extra effort.

Here is a simpler version that produces the same range error as the loop:

```
vector<int> v(5);
int x = v[5];
```

However, we doubt that you'd have considered that realistic and worth serious attention.

So what actually happens when we make such a range error? The subscript operation of `vector` knows the size of the `vector`, so it can check (and the `vector` we are using does; see §4.6 and §19.4). If that check fails, the subscript operation throws an exception of type `out_of_range`. So, if the off-by-one code above had been part of a program that caught exceptions, we would at least have gotten a decent error message:

```
int main()
try {
    vector<int> v;                                // a vector of ints
    for (int x; cin>>x; )
        v.push_back(x);                          // set values
    for (int i = 0; i<=v.size(); ++i)          // print values
        cout << "v[" << i << "] == " << v[i] << '\n';
} catch (out_of_range) {
    cerr << "Oops! Range error\n";
    return 1;
} catch (...) {                                    // catch all other exceptions
    cerr << "Exception: something went wrong\n";
    return 2;
}
```

Note that a range error is really a special case of the argument errors we discussed in §5.5.2. We didn't trust ourselves to consistently check the range of `vector` indices, so we told `vector`'s subscript operation to do it for us. For the reasons we outline, `vector`'s subscript function (called `vector::operator[]`) reports finding an error by throwing an exception. What else could it do? It has no idea what we would like to happen in case of a range error. The author of `vector` couldn't even know what programs his or her code would be part of.

5.6.3 Bad input

We'll postpone the detailed discussion of what to do with bad input until §10.6. However, once bad input is detected, it is dealt with using the same techniques and language features as argument errors and range errors. Here, we'll just show how you can tell if your input operations succeeded. Consider reading a floating-point number:

```
double d = 0;  
cin >> d;
```

We can test if the last input operation succeeded by testing `cin`:

```
if (cin) {  
    // all is well, and we can try reading again  
}  
else {  
    // the last read didn't succeed, so we take some other action  
}
```

There are several possible reasons for that input operation's failure. The one that should concern you right now is that there wasn't a `double` for `>>` to read.

During the early stages of development, we often want to indicate that we have found an error but aren't yet ready to do anything particularly clever about it; we just want to report the error and terminate the program. Later, maybe, we'll come back and do something more appropriate. For example:

```
double some_function()  
{  
    double d = 0;  
    cin >> d;  
    if (!cin) error("couldn't read a double in 'some_function()'");  
    // do something useful  
}
```

The condition `!cin` (“not `cin`,” that is, `cin` is not in a good state) means that the previous operation on `cin` failed.

The string passed to `error()` can then be printed as a help to debugging or as a message to the user. How can we write `error()` so as to be useful in a lot of programs? It can’t return a value because we wouldn’t know what to do with that value; instead `error()` is supposed to terminate the program after getting its message written. In addition, we might want to take some minor action before exiting, such as keeping a window alive long enough for us to read the message. That’s an obvious job for an exception (see §7.3).

The standard library defines a few types of exceptions, such as the `out_of_range` thrown by `vector`. It also supplies `runtime_error` which is pretty ideal for our needs because it holds a string that can be used by an error handler. So, we can write our simple `error()` like this:

```
void error(string s)
{
    throw runtime_error(s);
}
```

When we want to deal with `runtime_error` we simply catch it. For simple programs, catching `runtime_error` in `main()` is ideal:

```
int main()
try {
    // ... our program ...
    return 0;      // 0 indicates success
}
catch (runtime_error& e) {
    cerr << "runtime error: " << e.what() << '\n';
    keep_window_open();
    return 1;      // 1 indicates failure
}
```

The call `e.what()` extracts the error message from the `runtime_error`. The `&` in

```
catch(runtime_error& e) {
```

is an indicator that we want to “pass the exception by reference.” For now, please treat this as simply an irrelevant technicality. In §8.5.4–6, we explain what it means to pass something by reference.

Note that we used `cerr` rather than `cout` for our error output: `cerr` is exactly like `cout` except that it is meant for error output. By default both `cerr` and `cout`

write to the screen, but `cerr` isn't optimized so it is more resilient to errors, and on some operating systems it can be diverted to a different target, such as a file. Using `cerr` also has the simple effect of documenting that what we write relates to errors. Consequently, we use `cerr` for error messages.

As it happens, `out_of_range` is not a `runtime_error`, so catching `runtime_error` does not deal with the `out_of_range` errors that we might get from misuse of `vectors` and other standard library container types. However, both `out_of_range` and `runtime_error` are “exceptions,” so we can catch `exception` to deal with both:

```
int main()
try {
    // our program
    return 0;      // 0 indicates success
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    keep_window_open();
    return 1;      // 1 indicates failure
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    keep_window_open();
    return 2;      // 2 indicates failure
}
```

We added `catch(...)` to handle exceptions of any type whatsoever.

Dealing with exceptions of both type `out_of_range` and type `runtime_error` through a single type `exception`, said to be a common base (supertype) of both, is a most useful and general technique that we will explore in Chapters 13–16.

Note again that the return value from `main()` is passed to “the system” that invoked the program. Some systems (such as Unix) often use that value, whereas others (such as Windows) typically ignore it. A zero indicates successful completion and a nonzero return value from `main()` indicates some sort of failure.

When you use `error()`, you'll often wish to pass two pieces of information along to describe the problem. In that case, just concatenate the strings describing those two pieces of information. This is so common that we provide a second version of `error()` for that:

```
void error(string s1, string s2)
{
    throw runtime_error(s1+s2);
}
```

This simple error handling will do for a while, until our needs increase significantly and our sophistication as designers and programmers increases correspondingly. Note that we can use `error()` independently of how many function calls we have done on the way to the error: `error()` will find its way to the nearest catch of `runtime_error`, typically the one in `main()`. For examples of the use of exceptions and `error()`, see §7.3 and §7.7. If you don't catch an exception, you'll get a default system error (an “uncaught exception” error).

TRY THIS



To see what an uncaught exception error looks like, run a small program that uses `error()` without catching any exceptions.

5.6.4 Narrowing errors

In §3.9.2 we saw a nasty kind of error: when we assign a value that's “too large to fit” to a variable, it is implicitly truncated. For example:

```
int x = 2.9;
char c = 1066;
```

Here `x` will get the value `2` rather than `2.9`, because `x` is an `int` and `ints` don't have values that are fractions of an integer, just whole integers (obviously). Similarly, if we use the common ASCII character set, `c` will get the value `42` (representing the character `*`), rather than `1066`, because there is no `char` with the value `1066` in that character set.

In §3.9.2 we saw how we could protect ourselves against such narrowing by testing. Given exceptions (and templates; see §19.3) we can write a function that tests and throws a `runtime_error` exception if an assignment or initialization would lead to a changed value. For example:

```
int x1 = narrow_cast<int>(2.9);           // throws
int x2 = narrow_cast<int>(2.0);           // OK
char c1 = narrow_cast<char>(1066);         // throws
char c2 = narrow_cast<char>(85);           // OK
```

The `< . . . >` brackets are the same as are used for `vector<int>`. They are used when we need to specify a type, rather than a value, to express an idea. They are called *template arguments*. We can use `narrow_cast` when we need to convert a value and we are not sure “if it will fit”; it is defined in `std_lib_facilities.h` and implemented using `error()`. The word *cast* means “type conversion” and indicates the

operation's role in dealing with something that's broken (like a cast on a broken leg). Note that a cast doesn't change its operand; it produces a new value (of the type specified in the <...>) that corresponds to its operand value.

5.7 Logic errors

Once we have removed the initial compiler and linker errors, the program runs. Typically, what happens next is that no output is produced or that the output that the program produces is just wrong. This can occur for a number of reasons. Maybe your understanding of the underlying program logic is flawed; maybe you didn't write what you thought you wrote; or maybe you made some "silly error" in one of your **if**-statements, or whatever. Logic errors are usually the most difficult to find and eliminate, because at this stage the computer does what you asked it to. Your job now is to figure out why that wasn't really what you meant. Basically, a computer is a very fast moron. It does exactly what you tell it to do, and that can be most humbling.

Let us try to illustrate this with a simple example. Consider this code for finding the lowest, highest, and average temperature values in a set of data:

```
int main()
{
    vector<double> temps; // temperatures

    for (double temp; cin>>temp; )           // read and put into temps
        temps.push_back(temp);

    double sum = 0;
    double high_temp = 0;
    double low_temp = 0;

    for (int x : temps)
    {
        if(x > high_temp) high_temp = x;      // find high
        if(x < low_temp) low_temp = x;         // find low
        sum += x;                            // compute sum
    }

    cout << "High temperature: " << high_temp << '\n';
    cout << "Low temperature: " << low_temp << '\n';
    cout << "Average temperature: " << sum/temps.size() << '\n';
}
```

We tested this program by entering the hourly temperature values from the weather center in Lubbock, Texas, for February 16, 2004 (Texas still uses Fahrenheit):

```
-16.5, -23.2, -24.0, -25.7, -26.1, -18.6, -9.7, -2.4,  
7.5, 12.6, 23.8, 25.3, 28.0, 34.8, 36.7, 41.5,  
40.3, 42.6, 39.7, 35.4, 12.6, 6.5, -3.7, -14.3
```

The output was

```
High temperature: 42.6  
Low temperature: -26.1  
Average temperature: 9.3
```

A naive programmer would conclude that the program works just fine. An irresponsible programmer would ship it to a customer. It would be prudent to test it again with another set of data. This time use the temperatures from July 23, 2004:

```
76.5, 73.5, 71.0, 73.6, 70.1, 73.5, 77.6, 85.3,  
88.5, 91.7, 95.9, 99.2, 98.2, 100.6, 106.3, 112.4,  
110.2, 103.6, 94.9, 91.7, 88.4, 85.2, 85.4, 87.7
```

This time, the output was

```
High temperature: 112.4  
Low temperature: 0.0  
Average temperature: 89.2
```

Oops! Something is not quite right. Hard frost (0.0°F is about -18°C) in Lubbock in July would mean the end of the world! Did you spot the error? Since `low_temp` was initialized at `0.0`, it would remain `0.0` unless one of the temperatures in the data was below zero.

TRY THIS



Get this program to run. Check that our input really does produce that output. Try to “break” the program (i.e., get it to give wrong results) by giving it other input sets. What is the least amount of input you can give it to get it to fail?

Unfortunately, there are more errors in this program. What would happen if all of the temperatures were below zero? The initialization for `high_temp` has the

equivalent problem to `low_temp: high_temp` will remain at `0.0` unless there is a higher temperature in the data. This program wouldn't work for the South Pole in winter either.

These errors are fairly typical; they will not cause any errors when you compile the program or cause wrong results for “reasonable” inputs. However, we forgot to think about what we should consider “reasonable.” Here is an improved program:

```
int main()
{
    double sum = 0;
    double high_temp = -1000;           // initialize to impossibly low
    double low_temp = 1000;             // initialize to "impossibly high"
    int no_of_temps = 0;

    for (double temp; cin>>temp; ) {   // read temp
        ++no_of_temps;                 // count temperatures
        sum += temp;                  // compute sum
        if (temp > high_temp) high_temp = temp; // find high
        if (temp < low_temp) low_temp = temp; // find low
    }

    cout << "High temperature: " << high_temp << '\n';
    cout << "Low temperature: " << low_temp << '\n';
    cout << "Average temperature: " << sum/no_of_temps << '\n';
}
```

Does it work? How would you be certain? How would you precisely define “work”? Where did we get the values `1000` and `-1000`? Remember that we warned about “magic constants” (§5.5.1). Having `1000` and `-1000` as literal values in the middle of the program is bad style, but are the values also wrong? Are there places where the temperatures go below -1000°F (-573°C)? Are there places where the temperatures go above 1000°F (538°C)?

TRY THIS



Look it up. Check some information sources to pick good values for the `min_temp` (the “minimum temperature”) and `max_temp` (the “maximum temperature”) constants for our program. Those values will determine the limits of usefulness of our program.

5.8 Estimation

Imagine you have written a program that does a simple calculation, say, computing the area of a hexagon. You run it and it gives the area -34.56 . You just know that's wrong. Why? Because no shape has a negative area. So, you fix that bug (whatever it was) and get 21.65685 . Is that right? That's harder to say because we don't usually keep the formula for the area of a hexagon in our heads. What we must do before making fools of ourselves by delivering a program that produces ridiculous results is just to check that the answer is plausible. In this case, that's easy. A hexagon is much like a square. We scribble our regular hexagon on a piece of paper and eyeball it to be about the size of a 3-by-3 square. Such a square has the area 9. Bummer, our 21.65685 can't be right! So we work over our program again and get 10.3923 . Now, that just might be right!

The general point here has nothing to do with hexagons. The point is that unless we have some idea of what a correct answer will be like – even ever so approximately – we don't have a clue whether our result is reasonable. Always ask yourself this question:

1. Is this answer to this particular problem plausible?

You should also ask the more general (and often far harder) question:

2. How would I recognize a plausible result?

Here, we are not asking, “What’s the exact answer?” or “What’s the correct answer?” That’s what we are writing the program to tell us. All we want is to know that the answer is not ridiculous. Only when we know that we have a plausible answer does it make sense to proceed with further work.

Estimation is a noble art that combines common sense and some very simple arithmetic applied to a few facts. Some people are good at doing estimates in their heads, but we prefer scribbles “on the back of an envelope” because we find we get confused less often that way. What we call estimation here is an informal set of techniques that are sometimes (humorously) called *guesstimation* because they combine a bit of guessing with a bit of calculation.

TRY THIS



Our hexagon was regular with 2cm sides. Did we get that answer right? Just do the “back of the envelope” calculation. Take a piece of paper and scribble on it. Don’t feel that’s beneath you. Many famous scientists have been greatly admired for their ability to come up with an approximate answer using a pencil and the back of an envelope (or a napkin). This is an ability – a simple habit, really – that can save us a lot of time and confusion.

Often, making an estimate involves coming up with estimates of data that are needed for a proper calculation, but that we don't yet have. Imagine you have to test a program that estimates driving times between cities. Is a driving time of 15 hours and 33 minutes plausible for New York City to Denver? From London to Nice? Why or why not? What data do you have to "guess" to answer these questions? Often, a quick web search can be most helpful. For example, 2000 miles is not a bad guess on the road distance from New York City to Denver, and it would be hard (and illegal) to maintain an average speed of 130m/hr, so 15 hours is not plausible (15×130 is just a bit less than 2000). You can check: we overestimated both the distance and the average speed, but for a check of plausibility we don't have to be exactly right; we just have to guess well enough.

TRY THIS



Estimate those driving times. Also, estimate the corresponding flight times (using ordinary commercial air travel). Then, try to verify your estimates by using appropriate sources, such as maps and timetables. We'd use online sources.

5.9 Debugging

When you have written (drafted?) a program, it'll have errors. Small programs do occasionally compile and run correctly the first time you try. But if that happens for anything but a completely trivial program, you should at first be very, very suspicious. If it really did run correctly the first time, go tell your friends and celebrate – because this won't happen every year.

So, when you have written some code, you have to find and remove the errors. That process is usually called *debugging* and the errors *bugs*. The term *bug* is often claimed to have originated from a hardware failure caused by insects in the electronics in the days when computers were racks of vacuum tubes and relays filling rooms. Several people have been credited with the discovery and the application of the word *bug* to errors in software. The most famous of those is Grace Murray Hopper, the inventor of the COBOL programming language (§22.2.2.2). Whoever invented the term more than 50 years ago, *bug* is evocative and ubiquitous. The activity of deliberately searching for errors and removing them is called *debugging*.

Debugging works roughly like this:

1. Get the program to compile.
2. Get the program to link.
3. Get the program to do what it is supposed to do.

Basically, we go through this sequence again and again: hundreds of times, thousands of times, again and again for years for really large programs. Each time

something doesn't work we have to find what caused the problem and fix it. I consider debugging the most tedious and time-wasting aspect of programming and will go to great lengths during design and programming to minimize the amount of time spent hunting for bugs. Others find that hunt thrilling and the essence of programming – it can be as addictive as any video game and keep a programmer glued to the computer for days and nights (I can vouch for that from personal experience also).

Here is how *not* to debug:

```
while (the program doesn't appear to work) { // pseudo code
    Randomly look through the program for something that "looks odd"
    Change it to look better
}
```

Why do we bother to mention this? It's obviously a poor algorithm with little guarantee of success. Unfortunately, that description is only a slight caricature of what many people find themselves doing late at night when feeling particularly lost and clueless, having tried “everything else.”

The key question in debugging is

How would I know if the program actually worked correctly?

If you can't answer that question, you are in for a long and tedious debug session, and most likely your users are in for some frustration. We keep returning to this point because anything that helps answer that question minimizes debugging and helps produce correct and maintainable programs. Basically, we'd like to design our programs so that bugs have nowhere to hide. That's typically too much to ask for, but we aim to structure programs to minimize the chance of error and maximize the chance of finding the errors that do creep in.

5.9.1 Practical debug advice

Start thinking about debugging before you write the first line of code. Once you have a lot of code written it's too late to try to simplify debugging.

Decide how to report errors: “Use **error()** and catch **exception** in **main()**” will be your default answer in this book.

Make the program easy to read so that you have a chance of spotting the bugs:

- Comment your code well. That doesn't simply mean “Add a lot of comments.” You don't say in English what is better said in code. Rather, you say in the comments – as clearly and briefly as you can – what can't be said clearly in code:

- The name of the program
- The purpose of the program

- Who wrote this code and when
 - Version numbers
 - What complicated code fragments are supposed to do
 - What the general design ideas are
 - How the source code is organized
 - What assumptions are made about inputs
 - What parts of the code are still missing and what cases are still not handled
- Use meaningful names.
 - That doesn't simply mean "Use long names."
 - Use a consistent layout of code.
 - Your IDE tries to help, but it can't do everything and you are the one responsible.
 - The style used in this book is a reasonable starting point.
 - Break code into small functions, each expressing a logical action.
 - Try to avoid functions longer than a page or two; most functions will be much shorter.
 - Avoid complicated code sequences.
 - Try to avoid nested loops, nested **if**-statements, complicated conditions, etc. Unfortunately, you sometimes need those, but remember that complicated code is where bugs can most easily hide.
 - Use library facilities rather than your own code when you can.
 - A library is likely to be better thought out and better tested than what you could produce as an alternative while busily solving your main problem.

This is pretty abstract just now, but we'll show you example after example as we go along.

Get the program to compile. Obviously, your compiler is your best help here. Its error messages are usually helpful – even if we always wish for better ones – and, unless you are a real expert, assume that the compiler is always right; if you are a real expert, this book wasn't written for you. Occasionally, you will feel that the rules the compiler enforces are stupid and unnecessary (they rarely are) and that things could and ought to be simpler (indeed, but they are not). However,



as they say, “a poor craftsman curses his tools.” A good craftsman knows the strengths and weaknesses of his tools and adjusts his work accordingly. Here are some common compile-time errors:

- Is every string literal terminated?

```
cout << "Hello, << name << '\n';           // oops!
```

- Is every character literal terminated?

```
cout << "Hello, " << name << '\n';           // oops!
```

- Is every block terminated?

```
int f(int a)
{
    if (a>0) { /* do something */ else { /* do something else */ }
}
```

- Is every set of parentheses matched?

```
if (a<=0      // oops!
    x = f(y);
```

The compiler generally reports this kind of error “late”; it doesn’t know you meant to type a closing parenthesis after the **0**.

- Is every name declared?

- Did you include needed headers (for now, **#include "std_lib_facilities.h"**)?
- Is every name declared before it’s used?
- Did you spell all names correctly?

```
int count; /* ... */ ++Count;           // oops!
char ch;   /* ... */ Cin>>c;           // double oops!
```

- Did you terminate each expression statement with a semicolon?

```
x = sqrt(y)+2  // oops!
z = x+3;
```

We present more examples in this chapter’s drills. Also, keep in mind the classification of errors from §5.2.

After the program compiles and links, next comes what is typically the hardest part: figuring out why the program doesn’t do what it’s supposed to. You look at the output and try to figure out how your code could have produced that. Actually,

first you often look at a blank screen (or window), wondering how your program could have failed to produce any output. A common first problem with a Windows console-mode program is that the console window disappears before you have had a chance to see the output (if any). One solution is to call `keep_window_open()` from our `std_lib_facilities.h` at the end of `main()`. Then the program will ask for input before exiting and you can look at the output produced before giving it the input that will let it close the window.

When looking for a bug, carefully follow the code statement by statement from the last point that you are sure it was correct. Pretend you're the computer executing the program. Does the output match your expectations? Of course not, or you wouldn't be debugging.

- Often, when you don't see the problem, the reason is that you "see" what you expect to see rather than what you wrote. Consider:

```
for (int i = 0; i<=max; ++j) {           // oops! (twice)
    for (int i=0; 0<max; ++i);      // print the elements of v
        cout << "v[" << i << "]==" << v[i] << '\n';
    // ...
}
```

This last example came from a real program written by experienced programmers (we expect it was written very late some night).

- Often when you do not see the problem, the reason is that there is too much code being executed between the point where the program produced the last good output and the next output (or lack of output). Most programming environments provide a way to execute ("step through") the statements of a program one by one. Eventually, you'll learn to use such facilities, but for simple problems and simple programs, you can just temporarily put in a few extra output statements (using `cerr`) to help you see what's going on. For example:

```
int my_fct(int a, double d)
{
    int res = 0;
    cerr << "my_fct(" << a << "," << d << ")\\n";
    // ... misbehaving code here ...
    cerr << "my_fct() returns " << res << '\\n';
    return res;
}
```

- Insert statements that check invariants (that is, conditions that should always hold; see §9.4.3) in sections of code suspected of harboring bugs. For example:

```
int my_complicated_function(int a, int b, int c)
// the arguments are positive and a < b < c
{
    if (!(0 < a && a < b && b < c)) // ! means "not" and && means "and"
        error("bad arguments for mcf");
    // ...
}
```

- If that doesn't have any effect, insert invariants in sections of code not suspected of harboring bugs; if you can't find a bug, you are almost certainly looking in the wrong place.

A statement that states (asserts) an invariant is called an *assertion* (or just an *assert*).

Interestingly enough, there are many effective ways of programming. Different people successfully use dramatically different techniques. Many differences in debugging technique come from differences in the kinds of programs people work on; others seem to have to do with differences in the ways people think. To the best of our knowledge, there is no one best way to debug. One thing should always be remembered, though: messy code can easily harbor bugs. By keeping your code as simple, logical, and well formatted as possible, you decrease your debug time.

5.10 Pre- and post-conditions

Now, let us return to the question of how to deal with bad arguments to a function. The call of a function is basically the best point to think about correct code and to catch errors: this is where a logically separate computation starts (and ends on the return). Look at what we did in the piece of advice above:

```
int my_complicated_function(int a, int b, int c)
// the arguments are positive and a < b < c
{
    if (!(0 < a && a < b && b < c)) // ! means "not" and && means "and"
        error("bad arguments for mcf");
    // ...
}
```

First, we stated (in a comment) what the function required of its arguments, and then we checked that this requirement held (throwing an exception if it did not).

This is a good basic strategy. A requirement of a function upon its argument is often called a *pre-condition*: it must be true for the function to perform its action

correctly. The question is just what to do if the pre-condition is violated (doesn't hold). We basically have two choices:

1. Ignore it (hope/assume that all callers give correct arguments).
2. Check it (and report the error somehow).

Looking at it this way, argument types are just a way of having the compiler check the simplest pre-conditions for us and report them at compile time. For example:

```
int x = my_complicated_function(1, 2, "horsefeathers");
```

Here, the compiler will catch that the requirement ("pre-condition") that the third argument be an integer was violated. Basically, what we are talking about here is what to do with the requirements/pre-conditions that the compiler can't check.

Our suggestion is to always document pre-conditions in comments (so that a caller can see what a function expects). A function with no comments will be assumed to handle every possible argument value. But should we believe that callers read those comments and follow the rules? Sometimes we have to, but the "check the arguments in the callee" rule could be stated, "Let a function check its pre-conditions." We should do that whenever we don't see a reason not to. The reasons most often given for not checking pre-conditions are:

- Nobody would give bad arguments.
- It would slow down my code.
- It is too complicated to check.

The first reason can be reasonable only when we happen to know "who" calls a function – and in real-world code that can be very hard to know.

The second reason is valid far less often than people think and should most often be ignored as an example of "premature optimization." You can always remove checks if they really turn out to be a burden. You cannot easily gain the correctness they ensure or get back the nights' sleep you lost looking for bugs those tests could have caught.

The third reason is the serious one. It is easy (once you are an experienced programmer) to find examples where checking a pre-condition would take significantly more work than executing the function. An example is a lookup in a dictionary: a pre-condition is that the dictionary entries are sorted – and verifying that a dictionary is sorted can be far more expensive than a lookup. Sometimes, it can also be difficult to express a pre-condition in code and to be sure that you expressed it correctly. However, when you write a function, always consider if you can write a quick check of the pre-conditions, and do so unless you have a good reason not to.

Writing pre-conditions (even as comments) also has a significant benefit for the quality of your programs: it forces you to think about what a function re-

quires. If you can't state that simply and precisely in a couple of comment lines, you probably haven't thought hard enough about what you are doing. Experience shows that writing those pre-condition comments and the pre-condition tests helps you avoid many design mistakes. We did mention that we hated debugging; explicitly stating pre-conditions helps in avoiding design errors as well as catching usage errors early. Writing

```
int my_complicated_function(int a, int b, int c)
// the arguments are positive and a < b < c
{
    if (!(0 < a && a < b && b < c))      // ! means "not" and && means "and"
        error("bad arguments for mcf");
    // ...
}
```

saves you time and grief compared with the apparently simpler

```
int my_complicated_function(int a, int b, int c)
{
    // ...
}
```

5.10.1 Post-conditions

Stating pre-conditions helps us improve our design and catch usage errors early. Can this idea of explicitly stating requirements be used elsewhere? Yes, one more place immediately springs to mind: the return value! After all, we typically have to state what a function returns; that is, if we return a value from a function we are *always* making a promise about the return value (how else would a caller know what to expect?). Let's look at our area function (from §5.6.1) again:

```
// calculate area of a rectangle;
// throw a Bad_area exception in case of a bad argument
int area(int length, int width)
{
    if (length<=0 || width <=0) throw Bad_area();
    return length*width;
}
```

It checks its pre-condition, but it doesn't state it in the comment (that may be OK for such a short function) and it assumes that the computation is correct (that's

probably OK for such a trivial computation). However, we could be a bit more explicit:

```
int area(int length, int width)
// calculate area of a rectangle;
// pre-conditions: length and width are positive
// post-condition: returns a positive value that is the area
{
    if (length<=0 || width <=0) error("area() pre-condition");
    int a = length*width;
    if (a<=0) error("area() post-condition");
    return a;
}
```

We couldn't check the complete post-condition, but we checked the part that said that it should be positive.

TRY THIS



Find a pair of values so that the pre-condition of this version of area holds, but the post-condition doesn't.

Pre- and post-conditions provide basic sanity checks in code. As such they are closely connected to the notion of invariants (§9.4.3), correctness (§4.2, §5.2), and testing (Chapter 26).

5.11 Testing

How do we know when to stop debugging? Well, we keep debugging until we have found all the bugs – or at least we try to. How do we know that we have found the last bug? We don't. “The last bug” is a programmers' joke: there is no such creature; we never find “the last bug” in a large program. By the time we might have, we are busy modifying the program for some new use.



In addition to debugging we need a systematic way to search for errors. This is called *testing* and we'll get back to that in §7.3, the exercises in Chapter 10, and in Chapter 26. Basically, testing is executing a program with a large and systematically selected set of inputs and comparing the results to what was expected. A run with a given set of inputs is called a *test case*. Realistic programs can require millions of test cases. Basically, systematic testing cannot be done by humans typing in one test after another, so we'll have to wait a few chapters before we



have the tools necessary to properly approach testing. However, in the meantime, remember that we have to approach testing with the attitude that finding errors is good. Consider:

Attitude 1: I'm smarter than any program! I'll break that @#\$%^ code!

Attitude 2: I polished this code for two weeks. It's perfect!

Who do you think will find more errors? Of course, the very best is an experienced person with a bit of "attitude 1" who coolly, calmly, patiently, and systematically works through the possible failings of the program. Good testers are worth their weight in gold.

We try to be systematic in choosing our test cases and always try both correct and incorrect inputs. §7.3 gives the first example of this.



Drill

Below are 25 code fragments. Each is meant to be inserted into this "scaffolding":

```
#include "std_lib_facilities.h"

int main()
try {
    <<your code here>>
    keep_window_open();
    return 0;
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    keep_window_open();
    return 1;
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    keep_window_open();
    return 2;
}
```

Each has zero or more errors. Your task is to find and remove all errors in each program. When you have removed those bugs, the resulting program will compile, run, and write "Success!" Even if you think you have spotted an error, you still need to enter the (original, unimproved) program fragment and test it; you

may have guessed wrong about what the error is, or there may be more errors in a fragment than you spotted. Also, one purpose of this drill is to give you a feel for how your compiler reacts to different kinds of errors. Do not enter the scaffolding 25 times – that's a job for cut and paste or some similar “mechanical” technique. Do not fix problems by simply deleting a statement; repair them by changing, adding, or deleting a few characters.

```
1. Cout << "Success!\n";
2. cout << "Success!\n;
3. cout << "Success" << !\n"
4. cout << success << '\n';
5. string res = 7; vector<int> v(10); v[5] = res; cout << "Success!\n";
6. vector<int> v(10); v(5) = 7; if (v(5)!=7) cout << "Success!\n";
7. if (cond) cout << "Success!\n"; else cout << "Fail!\n";
8. bool c = false; if (c) cout << "Success!\n"; else cout << "Fail!\n";
9. string s = "ape"; boo c = "fool"<s; if (c) cout << "Success!\n";
10. string s = "ape"; if (s=="fool") cout << "Success!\n";
11. string s = "ape"; if (s=="fool") cout < "Success!\n";
12. string s = "ape"; if (s+"fool") cout < "Success!\n";
13. vector<char> v(5); for (int i=0; 0<v.size(); ++i) ; cout << "Success!\n";
14. vector<char> v(5); for (int i=0; i<=v.size(); ++i) ; cout << "Success!\n";
15. string s = "Success!\n"; for (int i=0; i<6; ++i) cout << s[i];
16. if (true) then cout << "Success!\n"; else cout << "Fail!\n";
17. int x = 2000; char c = x; if (c==2000) cout << "Success!\n";
18. string s = "Success!\n"; for (int i=0; i<10; ++i) cout << s[i];
19. vector v(5); for (int i=0; i<=v.size(); ++i) ; cout << "Success!\n";
20. int i=0; int j = 9; while (i<10) ++j; if (j<i) cout << "Success!\n";
21. int x = 2; double d = 5/(x-2); if (d==2*x+0.5) cout << "Success!\n";
22. string<char> s = "Success!\n"; for (int i=0; i<=10; ++i) cout << s[i];
23. int i=0; while (i<10) ++j; if (j<i) cout << "Success!\n";
24. int x = 4; double d = 5/(x-2); if (d=2*x+0.5) cout << "Success!\n";
25. cin << "Success!\n";
```

Review

1. Name four major types of errors and briefly define each one.
2. What kinds of errors can we ignore in student programs?
3. What guarantees should every completed project offer?
4. List three approaches we can take to eliminate errors in programs and produce acceptable software.
5. Why do we hate debugging?
6. What is a syntax error? Give five examples.
7. What is a type error? Give five examples.

8. What is a linker error? Give three examples.
9. What is a logic error? Give three examples.
10. List four potential sources of program errors discussed in the text.
11. How do you know if a result is plausible? What techniques do you have to answer such questions?
12. Compare and contrast having the caller of a function handle a run-time error vs. the called function's handling the run-time error.
13. Why is using exceptions a better idea than returning an “error value”?
14. How do you test if an input operation succeeded?
15. Describe the process of how exceptions are thrown and caught.
16. Why, with a `vector` called `v`, is `v[v.size()]` a range error? What would be the result of calling this?
17. Define *pre-condition* and *post-condition*; give an example (that is not the `area()` function from this chapter), preferably a computation that requires a loop.
18. When would you *not* test a pre-condition?
19. When would you *not* test a post-condition?
20. What are the steps in debugging a program?
21. Why does commenting help when debugging?
22. How does testing differ from debugging?

Terms

argument error	exception	requirement
assertion	invariant	run-time error
catch	link-time error	syntax error
compile-time error	logic error	testing
container	post-condition	throw
debugging	pre-condition	type error
error	range error	

Exercises

1. If you haven't already, do the **Try this** exercises from this chapter.
2. The following program takes in a temperature value in Celsius and converts it to Kelvin. This code has many errors in it. Find the errors, list them, and correct the code.

```
double ctok(double c)      // converts Celsius to Kelvin
{
    int k = c + 273.15;
    return int
}
```

```
int main()
{
    double c = 0;           // declare input variable
    cin >> d;             // retrieve temperature to input variable
    double k = ctok("c");   // convert temperature
    Cout << k << '\n' ;  // print out temperature
}
```

3. Absolute zero is the lowest temperature that can be reached; it is -273.15°C , or 0K. The above program, even when corrected, will produce erroneous results when given a temperature below this. Place a check in the main program that will produce an error if a temperature is given below -273.15°C .
4. Do exercise 3 again, but this time handle the error inside **ctok()**.
5. Add to the program so that it can also convert from Kelvin to Celsius.
6. Write a program that converts from Celsius to Fahrenheit and from Fahrenheit to Celsius (formula in §4.3.3). Use estimation (§5.8) to see if your results are plausible.
7. Quadratic equations are of the form

$$a \cdot x^2 + b \cdot x + c = 0$$

To solve these, one uses the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- There is a problem, though: if $b^2 - 4ac$ is less than zero, then it will fail. Write a program that can calculate x for a quadratic equation. Create a function that prints out the roots of a quadratic equation, given a , b , c . When the program detects an equation with no real roots, have it print out a message. How do you know that your results are plausible? Can you check that they are correct?
8. Write a program that reads and stores a series of integers and then computes the sum of the first N integers. First ask for N , then read the values into a **vector**, then calculate the sum of the first N values. For example:

“Please enter the number of values you want to sum:”

3

“Please enter some integers (press ‘|’ to stop):”

12 23 13 24 15 |

“The sum of the first **3** numbers (**12 23 13**) is **48**.”

- Handle all inputs. For example, make sure to give an error message if the user asks for a sum of more numbers than there are in the vector.
9. Modify the program from exercise 8 to write out an error if the result cannot be represented as an `int`.
 10. Modify the program from exercise 8 to use `double` instead of `int`. Also, make a `vector` of `doubles` containing the $N-1$ differences between adjacent values and write out that `vector` of differences.
 11. Write a program that writes out the first so many values of the Fibonacci series, that is, the series that starts with 1 1 2 3 5 8 13 21 34. The next number of the series is the sum of the two previous ones. Find the largest Fibonacci number that fits in an `int`.
 12. Implement a little guessing game called (for some obscure reason) “Bulls and Cows.” The program has a `vector` of four different integers in the range 0 to 9 (e.g., 1234 but not 1122) and it is the user’s task to discover those numbers by repeated guesses. Say the number to be guessed is 1234 and the user guesses 1359; the response should be “1 bull and 1 cow” because the user got one digit (1) right and in the right position (a bull) and one digit (3) right but in the wrong position (a cow). The guessing continues until the user gets four bulls, that is, has the four digits correct and in the correct order.
 13. The program is a bit tedious because the answer is hard-coded into the program. Make a version where the user can play repeatedly (without stopping and restarting the program) and each game has a new set of four digits. You can get four random digits by calling the random number generator `randint(10)` from `std_lib_facilities.h` four times. You will note that if you run that program repeatedly, it will pick the same sequence of four digits each time you start the program. To avoid that, ask the user to enter a number (any number) and call `strand(n)` where `n` is the number the user entered before calling `randint(10)`. Such an `n` is called a *seed*, and different seeds give different sequences of random numbers.
 14. Read (day-of-the-week,value) pairs from standard input. For example:

Tuesday 23 Friday 56 Tuesday –3 Thursday 99

Collect all the values for each day of the week in a `vector<int>`. Write out the values of the seven day-of-the-week `vectors`. Print out the sum of the values in each `vector`. Ignore illegal days of the week, such as `Funday`, but accept common synonyms such as `Mon` and `monday`. Write out the number of rejected values.

Postscript

Do you think we overemphasize errors? As novice programmers we would have thought so. The obvious and natural reaction is “It simply can’t be that bad!” Well, it is that bad. Many of the world’s best brains have been astounded and confounded by the difficulty of writing correct programs. In our experience, good mathematicians are the people most likely to underestimate the problem of bugs, but we all quickly exceed our natural capacity for writing programs that are correct the first time. You have been warned! Fortunately, after 50 years or so, we have a lot of experience in organizing code to minimize problems, and techniques to find the bugs that we – despite our best efforts – inevitably leave in our programs as we first write them. The techniques and examples in this chapter are a good start.





Writing a Program

“Programming is understanding.”

– Kristen Nygaard

Writing a program involves gradually refining your ideas of what you want to do and how you want to express it. In this chapter and the next, we will develop a program from a first vague idea through stages of analysis, design, implementation, testing, redesign, and re-implementation. Our aim is to give you some idea of the kind of thinking that goes on when you develop a piece of code. In the process, we discuss program organization, user-defined types, and input processing.

6.1 A problem	6.5 Turning a grammar into code
6.2 Thinking about the problem	6.5.1 Implementing grammar rules
6.2.1 Stages of development	6.5.2 Expressions
6.2.2 Strategy	6.5.3 Terms
6.3 Back to the calculator!	6.5.4 Primary expressions
6.3.1 First attempt	6.6 Trying the first version
6.3.2 Tokens	6.7 Trying the second version
6.3.3 Implementing tokens	6.8 Token streams
6.3.4 Using tokens	6.8.1 Implementing <code>Token_stream</code>
6.3.5 Back to the drawing board	6.8.2 Reading tokens
6.4 Grammars	6.8.3 Reading numbers
6.4.1 A detour: English grammar	6.9 Program structure
6.4.2 Writing a grammar	

6.1 A problem



Writing a program starts with a problem; that is, you have a problem that you'd like a program to help solve. Understanding that problem is key to a good program. After all, a program that solves the wrong problem is likely to be of little use to you, however elegant it may be. There are happy accidents when a program just happens to be useful for something for which it was never intended, but let's not rely on such rare luck. What we want is a program that simply and cleanly solves the problem we decided to solve.

At this stage, what would be a good program to look at? A program that

- Illustrates design and programming techniques
- Gives us a chance to explore the kinds of decisions that a programmer must make and the considerations that go into such decisions
- Doesn't require too many new programming language constructs
- Is complicated enough to require thought about its design
- Allows for many variations in its solution
- Solves an easily understood problem
- Solves a problem that's worth solving
- Has a solution that is small enough to completely present and completely comprehend

We chose “Get the computer to do ordinary arithmetic on expressions we type in”; that is, we want to write a simple calculator. Such programs are clearly useful; every desktop computer comes with such a program, and you can even buy computers specially built to run nothing but such programs: pocket calculators.

For example, if you enter

2+3.1*4

the program should respond

14.4

Unfortunately, such a calculator program doesn't give us anything we don't already have available on our computer, but that would be too much to ask from a first program.

6.2 Thinking about the problem

So how do we start? Basically, think a bit about the problem and how to solve it. First think about what the program should do and how you'd like to interact with it. Later, you can think about how the program could be written to do that. Try writing down a brief sketch of an idea for a solution, and see what's wrong with that first idea. Maybe discuss the problem and how to solve it with a friend. Trying to explain something to a friend is a marvelous way of figuring out what's wrong with ideas, even better than writing them down; paper (or a computer) doesn't talk back at you and challenge your assumptions. Ideally, design isn't a lonely activity.

Unfortunately, there isn't a general strategy for problem solving that works for all people and all problems. There are whole books that claim to help you be better at problem solving and another huge branch of literature that deals with program design. We won't go there. Instead, we'll present a page's worth of suggestions for a general strategy for the kind of smaller problems an individual might face. After that, we'll quickly proceed to try out these suggestions on our tiny calculator problem.

When reading our discussion of the calculator program, we recommend that you adopt a more than usually skeptical attitude. For realism, we evolve our program through a series of versions, presenting the reasoning that leads to each version along the way. Obviously, much of that reasoning must be incomplete or even faulty, or we would finish the chapter early. As we go along, we provide examples of the kinds of concerns and reasoning that designers and programmers deal with all the time. We don't reach a version of the program that we are happy with until the end of the next chapter.

Please keep in mind that for this chapter and the next, the way we get to the final version of the program – the journey through partial solutions, ideas, and mistakes – is at least as important as that final version and more important than the language-technical details we encounter along the way (we will get back to those later).

6.2.1 Stages of development

Here is a bit of terminology for program development. As you work on a problem you repeatedly go through these stages:

- *Analysis:* Figure out what should be done and write a description of your (current) understanding of that. Such a description is called a *set of requirements* or a *specification*. We will not go into details about how such requirements are developed and written down. That's beyond the scope of this book, but it becomes increasingly important as the size of problems increases.
- *Design:* Create an overall structure for the system, deciding which parts the implementation should have and how those parts should communicate. As part of the design consider which tools – such as libraries – can help you structure the program.
- *Implementation:* Write the code, debug it, and test that it actually does what it is supposed to do.

6.2.2 Strategy

Here are some suggestions that – when applied thoughtfully and with imagination – help with many programming projects:

- What is the problem to be solved? The first thing to do is to try to be specific about what you are trying to accomplish. This typically involves constructing a description of the problem or – if someone else gave you such a statement – trying to figure out what it really means. At this point you should take the user's point of view (not the programmer/implementation's view); that is, you should ask questions about what the program should do, not about how it is going to do it. Ask: "What can this program do for me?" and "How would I like to interact with this program?" Remember, most of us have lots of experience as users of computers on which to draw.
 - Is the problem statement clear? For real problems, it never is. Even for a student exercise, it can be hard to be sufficiently precise and specific. So we try to clarify it. It would be a pity if we solved the wrong problem. Another pitfall is to ask for too much. When we try to figure out what we want, we easily get too greedy/ambitious. It is almost always better to ask for less to make a program easier to specify, easier to understand, easier to use, and (hopefully) easier to implement. Once it works, we can always build a fancier "version 2.0" based on our experience.

- Does the problem seem manageable, given the time, skills, and tools available? There is little point in starting a project that you couldn't possibly complete. If there isn't sufficient time to implement (including testing) a program that does all that is required, it is usually wise not to start. Instead, acquire more resources (especially more time) or (best of all) modify the requirements to simplify your task.
- Try breaking the program into manageable parts. Even the smallest program for solving a real problem is large enough to be subdivided.
 - Do you know of any tools, libraries, etc. that might help? The answer is almost always yes. Even at the earliest stage of learning to program, you have parts of the C++ standard library. Later, you'll know large parts of that standard library and how to find more. You'll have graphics and GUI libraries, a matrix library, etc. Once you have gained a little experience, you will be able to find thousands of libraries by simple web searches. Remember: There is little value in reinventing the wheel when you are building software for real use. When learning to program it is a different matter; then, reinventing the wheel to see how that is done is often a good idea. Any time you save by using a good library can be spent on other parts of your problem, or on rest. How do you know that a library is appropriate for your task and of sufficient quality? That's a hard problem. Part of the solution is to ask colleagues, to ask in discussion groups, and to try small examples before committing to use a library.
 - Look for parts of a solution that can be separately described (and potentially used in several places in a program or even in other programs). To find such parts requires experience, so we provide many examples throughout this book. We have already used `vector`, `string`, and `iostreams` (`cin` and `cout`). This chapter gives the first complete examples of design, implementation, and use of program parts provided as user-defined types (`Token` and `Token_stream`). Chapters 8 and 13–15 present many more examples together with their design rationales. For now, consider an analogy: If we were to design a car, we would start by identifying parts, such as wheels, engine, seats, door handles, etc., on which we could work separately before assembling the complete car. There are tens of thousands of such parts of a modern car. A real-world program is no different in that respect, except of course that the parts are code. We would not try to build a car directly out of raw materials, such as iron, plastics, and wood. Nor would we try to build a major program directly out of (just) the expressions, statements, and types provided by the language. Designing and implementing such

parts is a major theme of this book and of software development in general; see the discussions of user-defined types (Chapter 9), class hierarchies (Chapter 14), and generic types (Chapter 20).

- Build a small, limited version of the program that solves a key part of the problem. When we start, we rarely know the problem well. We often think we do (don't we know what a calculator program is?), but we don't. Only a combination of thinking about the problem (analysis) and experimentation (design and implementation) gives us the solid understanding that we need to write a good program. So, we build a small, limited version
 - To bring out problems in our understanding, ideas, and tools.
 - To see if details of the problem statement need changing to make the problem manageable. It is rare to find that we had anticipated everything when we analyzed the problem and made the initial design. We should take advantage of the feedback that writing code and testing give us.

Sometimes, such a limited initial version aimed at experimentation is called a *prototype*. If (as is likely) our first version doesn't work or is so ugly and awkward that we don't want to work with it, we throw it away and make another limited version based on our experience. Repeat until we find a version that we are happy with. Do not proceed with a mess; messes just grow with time.

- Build a full-scale solution, ideally by using parts of the initial version. The ideal is to grow a program from working parts rather than writing all the code at once. The alternative is to hope that by some miracle an untested idea will work and do what we want.

6.3 Back to the calculator!

How do we want to interact with the calculator? That's easy: we know how to use **cin** and **cout**, but graphical user interfaces (GUIs) are not explained until Chapter 16, so we'll stick to the keyboard and a console window. Given expressions as input from the keyboard, we evaluate them and write out the resulting value to the screen. For example:

```
Expression: 2+2
Result: 4
Expression: 2+2*3
Result: 8
Expression: 2+3-25/5
Result: 0
```

The expressions, e.g., **2+2** and **2+2*3**, should be entered by the user; the rest is produced by the program. We chose to output **Expression:** to prompt the user. We could have chosen **Please enter an expression followed by a newline** but that seemed verbose and pointless. On the other hand, a pleasantly short prompt, such as **>**, seemed too cryptic. Sketching out such examples of use early on is important. They provide a very practical definition of what the program should minimally do. When discussing design and analysis, such examples of use are called *use cases*.

When faced with the calculator problem for the first time, most people come up with a first idea like this for the main logic of the program:

```
read_a_line  
calculate    // do the work  
write_result
```

This kind of “scribbles” clearly isn’t code; it’s called *pseudo code*. We tend to use it in the early stages of design when we are not yet certain exactly what our notation means. For example, is “calculate” a function call? If so, what would be its arguments? It is simply too early to answer such questions.

6.3.1 First attempt

At this point, we are not really ready to write the calculator program. We simply haven’t thought hard enough, but thinking is hard work and – like most programmers – we are anxious to write some code. So let’s take a chance, write a simple calculator, and see where it leads us. The first idea is something like

```
#include "std_lib_facilities.h"  
  
int main()  
{  
    cout << "Please enter expression (we can handle + and -): ";  
    int lval = 0;  
    int rval;  
    char op;  
    int res;  
    cin >> lval >> op >> rval;    // read something like 1 + 3  
  
    if (op == '+')  
        res = lval + rval;    // addition  
    else if (op == '-')  
        res = lval - rval;    // subtraction  
  
    cout << "Result: " << res << '\n';  
    keep_window_open();
```

```
    return 0;
}
```

That is, read a pair of values separated by an operator, such as **2+2**, compute the result (in this case **4**), and print the resulting value. We chose the variable names **lval** for left-hand value and **rval** for right-hand value.

This (sort of) works! So what if this program isn't quite complete? It feels great to get something running! Maybe this programming and computer science stuff is easier than the rumors say. Well, maybe, but let's not get too carried away by an early success. Let's

1. Clean up the code a bit
2. Add multiplication and division (e.g., **2*3**)
3. Add the ability to handle more than one operand (e.g., **1+2+3**)

In particular, we know that we should always check that our input is reasonable (in our hurry, we "forgot") and that testing a value against many constants is best done by a **switch**-statement rather than an **if**-statement.

The "chaining" of operations, such as **1+2+3+4**, we will handle by adding the values as they are read; that is, we start with **1**, see **+2** and add **2** to **1** (getting an intermediate result **3**), see **+3** and add that **3** to our intermediate result (**3**), and so on. After a few false starts and after correcting a few syntax and logic errors, we get

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Please enter expression (we can handle +, -, *, and /)\n";
    cout << "add an x to end expression (e.g., 1+2*3x): ";
    int lval = 0;
    int rval;
    cin >> lval;                                // read leftmost operand
    if (!cin) error("no first operand");
    for (char op; cin >> op; ) {                // read operator and right-hand operand
        // repeatedly
        if (op != 'x') cin >> rval;
        if (!cin) error("no second operand");
        switch(op) {
            case '+':
                lval += rval;                  // add: lval = lval + rval
                break;
            case '-':
                lval -= rval;                  // subtract: lval = lval - rval
                break;
        }
    }
}
```

```

        case '*':
            lval *= rval;      // multiply: lval = lval * rval
            break;
        case '/':
            lval /= rval;      // divide: lval = lval / rval
            break;
        default:             // not another operator: print result
            cout << "Result: " << lval << '\n';
            keep_window_open();
            return 0;
        }
    }
error("bad expression");
}

```

This isn't bad, but then we try **1+2*3** and see that the result is **9** and not the **7** our arithmetic teachers told us was the right answer. Similarly, **1-2*3** gives **-3** rather than the **-5** we expected. We are doing the operations in the wrong order: **1+2*3** is calculated as **(1+2)*3** rather than as the conventional **1+(2*3)**. Similarly, **1-2*3** is calculated as **(1-2)*3** rather than as the conventional **1-(2*3)**. Bummer! We might consider the convention that “multiplication binds tighter than addition” as a silly old convention, but hundreds of years of convention will not disappear just to simplify our programming.

6.3.2 Tokens

So (somehow), we have to “look ahead” on the line to see if there is a ***** (or a **/**). If so, we have to (somehow) adjust the evaluation order from the simple and obvious left-to-right order. Unfortunately, trying to barge ahead here, we immediately hit a couple of snags:

1. We don't actually require an expression to be on one line. For example:

```

1
+
2

```

works perfectly with our code so far.

2. How do we search for a ***** (or a **/**) among digits, plusses, minuses, and parentheses on several input lines?
3. How do we remember where a ***** was?
4. How do we handle evaluation that's not strictly left-to-right (e.g., **1+2*3**)?

Having decided to be super-optimists, we'll solve problems 1–3 first and not worry about 4 until later.

Also, we'll ask around for help. Surely someone will know a conventional way of reading “stuff,” such as numbers and operators, from input and storing it in a way that lets us look at it in convenient ways. The conventional and very useful answer is “tokenize”: first input characters are read and assembled into *tokens*, so if you type in

45+11.5/7

the program should produce a list of tokens representing

**45
+
11.5
/
7**

A *token* is a sequence of characters that represents something we consider a unit, such as a number or an operator. That's the way a C++ compiler deals with its source. Actually, “tokenizing” in some form or another is the way most analysis of text starts. Following the example of C++ expression, we see the need for three kinds of tokens:

- Floating-point-literals: as defined by C++, e.g., **3.14**, **0.274e2**, and **42**
- Operators: e.g., **+**, **-**, *****, **/**, **%**
- Parentheses: **(**, **)**

The floating-point-literals look as if they may become a problem: reading **12** seems much easier than reading **12.3e-3**, but calculators do tend to do floating-point arithmetic. Similarly, we suspect that we'll have to accept parentheses to have our calculator deemed useful.

How do we represent such tokens in our program? We could try to keep track of where each token started (and ended), but that gets messy (especially if we allow expressions to span line boundaries). Also, if we keep a number as a string of characters, we later have to figure out what its value is; that is, if we see **42** and store the characters **4** and **2** somewhere, we then later have to figure out that those characters represent the numerical value **42** (i.e., **4*10+2**). The obvious – and conventional – solution is to represent each token as a *(kind,value)* pair. The *kind* tells us if a token is a number, an operator, or a parenthesis. For a number, and in this example only for a number, we use its numerical value as its *value*.

So how do we express the idea of a *(kind,value)* pair in code? We define a type **Token** to represent tokens. Why? Remember why we use types: they hold the data we need and give us useful operations on that data. For example, **ints** hold integers and give us addition, subtraction, multiplication, division, and remainder,

whereas **strings** hold sequences of characters and give us concatenation and subscripting. The C++ language and its standard library give us many types such as **char**, **int**, **double**, **string**, **vector**, and **ostream**, but not a **Token** type. In fact, there is a huge number of types – thousands or tens of thousands – that we would like to have, but the language and its standard library do not supply them. Among our favorite types that are not supported are **Matrix** (see Chapter 24), **Date** (see Chapter 9), and infinite precision integers (try searching the web for “**Bignum**”). If you think about it for a second, you’ll realize that a language cannot supply tens of thousands of types: who would define them, who would implement them, how would you find them, and how thick would the manual have to be? Like most modern languages, C++ escapes that problem by letting us define our own types (*user-defined types*) when we need them.

6.3.3 Implementing tokens

What should a token look like in our program? In other words, what would we like our **Token** type to be? A **Token** must be able to represent operators, such as **+** and **-**, and numeric values, such as **42** and **3.14**. The obvious implementation is something that can represent what “kind” a token is and hold the numeric value for tokens that have one:

Token:	Token:
kind: plus	kind: number
value:	value: 3.14

There are many ways that this idea could be represented in C++ code. Here is the simplest that we found useful:

```
class Token {      // a very simple user-defined type
public:
    char kind;
    double value;
};
```

A **Token** is a type (like **int** or **char**), so it can be used to define variables and hold values. It has two parts (called *members*): **kind** and **value**. The keyword **class** means “user-defined type”; it indicates that a type with zero or more members is being defined. The first member, **kind**, is a character, **char**, so that it conveniently can hold **'+'** and **'*'** to represent **+** and *****. We can use it to make types like this:

```
Token t;          // t is a Token
t.kind = '+';    // t represents a +
```

```
Token t2;           // t2 is another Token
t2.kind = '8';     // we use the digit 8 as the "kind" for numbers
t2.value = 3.14;
```

We use the member access notation, `object_name . member_name`, to access a member. You can read `t.kind` as “`t`’s `kind`” and `t2.value` as “`t2`’s `value`.” We can copy **Tokens** just as we can copy **ints**:

```
Token tt = t;       // copy initialization
if (tt.kind != t.kind) error("impossible!");
t = t2;             // assignment
cout << t.value;    // will print 3.14
```

Given **Token**, we can represent the expression `(1.5+4)*11` using seven tokens like this:

'('	'8'	'+'	'8')'	'*'	'8'
1.5			4			11

Note that for simple tokens, such as `+`, we don’t need the value, so we don’t use its `value` member. We needed a character to mean “number” and picked `'8'` just because `'8'` obviously isn’t an operator or a punctuation character. Using `'8'` to mean “number” is a bit cryptic, but it’ll do for now.

Token is an example of a C++ user-defined type. A user-defined type can have member functions (operations) as well as data members. There can be many reasons for defining member functions. Here, we’ll just provide two member functions to give us a more convenient way of initializing **Tokens**:

```
class Token {
public:
    char kind;      // what kind of token
    double value;   // for numbers: a value
};
```

We can now initialize (“construct”) **Token** objects. For example:

```
Token t1 {'+'};      // initialize t1 so that t1.kind = '+'
Token t2 {'8',11.5}; // initialize t2 so that t2.kind = '8' and t2.value = 11.5
```

For more about initializing class objects, see §9.4.2 and §9.7.

6.3.4 Using tokens

So, maybe now we can complete our calculator! However, maybe a small amount of planning ahead would be worthwhile. How would we use **Tokens** in the calculator? We can read input into a **vector** of **Tokens**:

```
Token get_token();    // function to read a token from cin

vector<Token> tok; // we'll put the tokens here

int main()
{
    while (cin) {
        Token t = get_token();
        tok.push_back(t);
    }
    // ...
}
```

Now we can read an expression first and evaluate later. For example, for **11*12**, we get

'8'	'*'	'8'
11		12

We can look at that to find the multiplication and its operands. Having done that, we can easily perform the multiplication because the numbers 11 and 12 are stored as numeric values and not as strings.

Now let's look at more complex expressions. Given **1+2*3**, **tok** will contain five **Tokens**:

'8'	'+'	'8'	'*'	'8'
1		2		3

Now we could find the multiply operation by a simple loop:

```
for (int i = 0; i<tok.size(); ++i) {
    if (tok[i].kind=='*') {           // we found a multiply!
        double d = tok[i-1].value*tok[i+1].value;
        // now what?
    }
}
```

Yes, but now what? What do we do with that product **d**? How do we decide in which order to evaluate the sub-expressions? Well, **+** comes before ***** so we can't just evaluate from left to right. We could try right-to-left evaluation! That would work for **1+2*3** but not for **1*2+3**. Worse still, consider **1+2*3+4**. This example has to be evaluated “inside out”: **1+(2*3)+4**. And how will we handle parentheses, as we eventually will have to do? We seem to have hit a dead end. We need to back off, stop programming for a while, and think about how we read and understand an input string and evaluate it as an arithmetic expression.

So, this first enthusiastic attempt to solve the problem (writing a calculator) ran out of steam. That's not uncommon for first tries, and it serves the important role of helping us understand the problem. In this case, it even gave us the useful notion of a token, which itself is an example of the notion of a *(name,value)* pair that we will encounter again and again. However, we must always make sure that such relatively thoughtless and unplanned “coding” doesn't steal too much time. We should do very little programming before we have done at least a bit of analysis (understanding the problem) and design (deciding on an overall structure of a solution).

TRY THIS



On the other hand, why shouldn't we be able to find a simple solution to this problem? It doesn't seem to be all that difficult. If nothing else, trying would give us a better appreciation of the problem and the eventual solution. Consider what you might do right away. For example, look at the input **12.5+2**. We could tokenize that, decide that the expression was simple, and compute the answer. That may be a bit messy, but straightforward, so maybe we could proceed in this direction and find something that's good enough! Consider what to do if we found both a **+** and a ***** in the line **2+3*4**. That too can be handled by “brute force.” How would we deal with a complicated expression, such as **1+2*3/4%5+(6-7*(8))**? And how would we deal with errors, such as **2+*3** and **2&3**? Consider this for a while, maybe doodling a bit on a piece of paper trying to outline possible solutions and interesting or important input expressions.

6.3.5 Back to the drawing board

Now, we will look at the problem again and try not to dash ahead with another half-baked solution. One thing that we did discover was that having the program (calculator) evaluate only a single expression was tedious. We would like to be

able to compute several expressions in a single invocation of our program; that is, our pseudo code grows to

```
while (not_finished) {
    read_a_line
    calculate           // do the work
    write_result
}
```

Clearly this is a complication, but when we think about how we use calculators, we realize that doing several calculations is very common. Could we let the user invoke our program several times to do several calculations? We could, but program startup is unfortunately (and unreasonably) slow on many modern operating systems, so we'd better not rely on that.

As we look at this pseudo code, our early attempts at solutions, and our examples of use, several questions – some with tentative answers – arise:

1. If we type in **45+5/7**, how do we find the individual parts **45**, **+**, **5**, **/**, and **7** in the input? (Tokenize!)
2. What terminates an input expression? A newline, of course! (Always be suspicious of “of course”: “of course” is not a reason.)
3. How do we represent **45+5/7** as data so that we can evaluate it? Before doing the addition we must somehow turn the characters **4** and **5** into the integer value **45** (i.e., **4*10+5**). (So tokenizing is part of the solution.)
4. How do we make sure that **45+5/7** is evaluated as **45+(5/7)** and not as **(45+5)/7?**
5. What's the value of **5/7**? About **.71**, but that's not an integer. Based on experience with calculators, we know that people would expect a floating-point result. Should we also allow floating-point inputs? Sure!
6. Can we have variables? For example, could we write

```
v=7
m=9
v*m
```

Good idea, but let's wait until later. Let's first get the basics working.

Possibly the most important decision here is the answer to question 6. In §7.8, you'll see that if we had said yes we'd have almost doubled the size of the initial project. That would have more than doubled the time needed to get the initial version running. Our guess is that if you really are a novice, it would have at least



quadrupled the effort needed and most likely pushed the project beyond your patience. It is most important to avoid “feature creep” early in a project. Instead, always first build a simple version, implementing the essential features only. Once you have something running, you can get more ambitious. It is far easier to build a program in stages than all at once. Saying yes to question 6 would have had yet another bad effect: it would have made it hard to resist the temptation to add further “neat features” along the line. How about adding the usual mathematical functions? How about adding loops? Once we start adding “neat features” it is hard to stop.

From a programmer’s point of view, questions 1, 3, and 4 are the most bothersome. They are also related, because once we have found a **45** or a **+**, what do we do with them? That is, how do we store them in our program? Obviously, tokenizing is part of the solution, but only part.

What would an experienced programmer do? When we are faced with a tricky technical question, there often is a standard answer. We know that people have been writing calculator programs for at least as long as there have been computers taking symbolic input from a keyboard. That is at least for 50 years. There has to be a standard answer! In such a situation, the experienced programmer consults colleagues and/or the literature. It would be silly to barge on, hoping to beat 50 years of experience in a morning.

6.4 Grammars

There is a standard answer to the question of how to make sense of expressions: first input characters are read and assembled into tokens (as we discovered). So if you type in

45+11.5/7

the program should produce a list of tokens representing

45
+
11.5
/
7

A token is a sequence of characters that represents something we consider a unit, such as a number or an operator.

After tokens have been produced, the program must ensure that complete expressions are understood correctly. For example, we know that **45+11.5/7** means

45+(11.5/7) and not **(45+11.5)/7**, but how do we teach the program that useful rule (division “binds tighter” than addition)? The standard answer is that we write a grammar defining the syntax of our input and then write a program that implements the rules of that grammar. For example:

// a simple expression grammar:

Expression:

Term

Expression "+" Term

// addition

Expression "-" Term

// subtraction

Term:

Primary

Term "*" Primary

// multiplication

Term "/" Primary

// division

Term "%" Primary

// remainder (modulo)

Primary:

Number

"(" Expression ")"

// grouping

Number:

floating-point-literal

This is a set of simple rules. The last rule is read “A **Number** is a **floating-point-literal**.” The next-to-last rule says, “A **Primary** is a **Number** or '(' followed by an **Expression** followed by ')'.” The rules for **Expression** and **Term** are similar; each is defined in terms of one of the rules that follow.

As seen in §6.3.2, our tokens – as borrowed from the C++ definition – are

- **floating-point-literal** (as defined by C++, e.g., **3.14**, **0.274e2**, or **42**)
- **+, -, *, /, %** (the operators)
- **(,)** (the parentheses)

From our first tentative pseudo code to this approach, using tokens and a grammar is actually a huge conceptual jump. It’s the kind of jump we hope for but rarely manage without help. This is what experience, the literature, and Mentors are for.

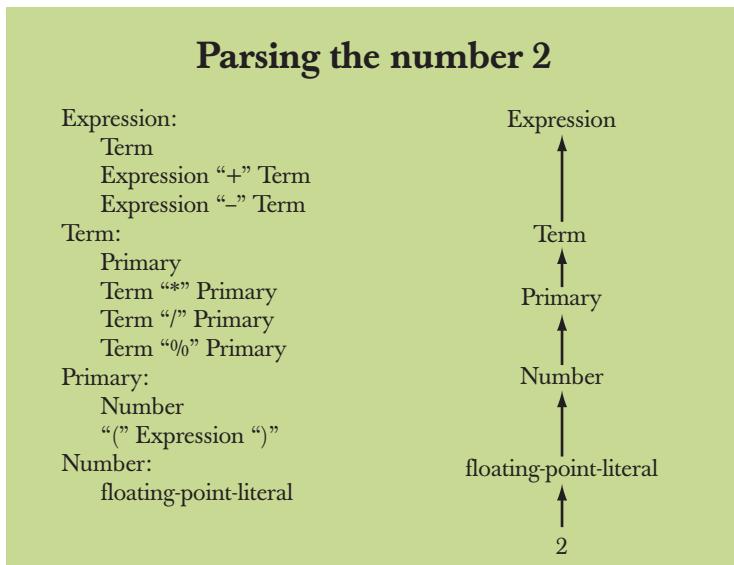
At first glance, a grammar probably looks like complete nonsense. Technical notation often does. However, please keep in mind that it is a general and elegant (as you will eventually appreciate) notation for something you have been able to do since middle school (or earlier). You have no problem calculating **1-2*3** and **1+2-3** and **3*2+4/2**. It seems hardwired in your brain. However, could you explain how you do it? Could you explain it well enough for someone who had never seen conventional arithmetic to grasp? Could you do so for every combination of

operators and operands? To articulate an explanation in sufficient detail and precisely enough for a computer to understand, we need a notation – and a grammar is a most powerful and conventional tool for that.

How do you read a grammar? Basically, given some input, you start with the “top rule,” **Expression**, and search through the rules to find a match for the tokens as they are read. Reading a stream of tokens according to a grammar is called *parsing*, and a program that does that is often called a *parser* or a *syntax analyzer*. Our parser reads the tokens from left to right, just like we type them and read them. Let’s try something really simple: Is **2** an expression?

1. An **Expression** must be a **Term** or end with a **Term**. That **Term** must be a **Primary** or end with a **Primary**. That **Primary** must start with a **(** or be a **Number**. Obviously, **2** is not a **(**, but a **floating-point-literal**, which is a **Number**, which is a **Primary**.
2. That **Primary** (the **Number 2**) isn’t preceded by a **/**, *****, or **%**, so it is a complete **Term** (rather than the end of a **/**, *****, or **%** expression).
3. That **Term** (the **Primary 2**) isn’t preceded by a **+** or **-**, so it is a complete **Expression** (rather than the end of a **+** or **-** expression).

So yes, according to our grammar, **2** is an expression. We can illustrate the progression through the grammar like this:

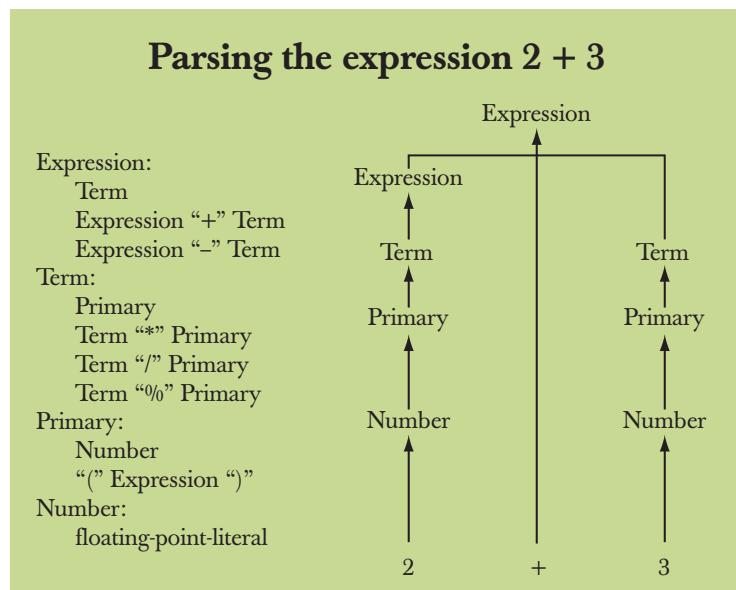


This represents the path we followed through the definitions. Retracing our path, we can say that **2** is an **Expression** because **2** is a **floating-point-literal**, which is a **Number**, which is a **Primary**, which is a **Term**, which is an **Expression**.

Let's try something a bit more complicated: Is **2+3** an **Expression**? Naturally, much of the reasoning is the same as for **2**:

1. An **Expression** must be a **Term** or end with a **Term**, which must be a **Primary** or end with a **Primary**, and a **Primary** must start with a **(** or be a **Number**. Obviously **2** is not a **(**, but it is a **floating-point-literal**, which is a **Number**, which is a **Primary**.
2. That **Primary** (the **Number 2**) isn't preceded by a **/**, *****, or **%**, so it is a complete **Term** (rather than the end of a **/**, *****, or **%** expression).
3. That **Term** (the **Primary 2**) is followed by a **+**, so it is the end of the first part of an **Expression** and we must look for the **Term** after the **+**. In exactly the same way as we found that **2** was a **Term**, we find that **3** is a **Term**. Since **3** is not followed by a **+** or a **-** it is a complete **Term** (rather than the first part of a **+** or **- Expression**). Therefore, **2+3** matches the **Expression+Term** rule and is an **Expression**.

Again, we can illustrate this reasoning graphically (leaving out the **floating-point-literal** to **Number** rule to simplify):



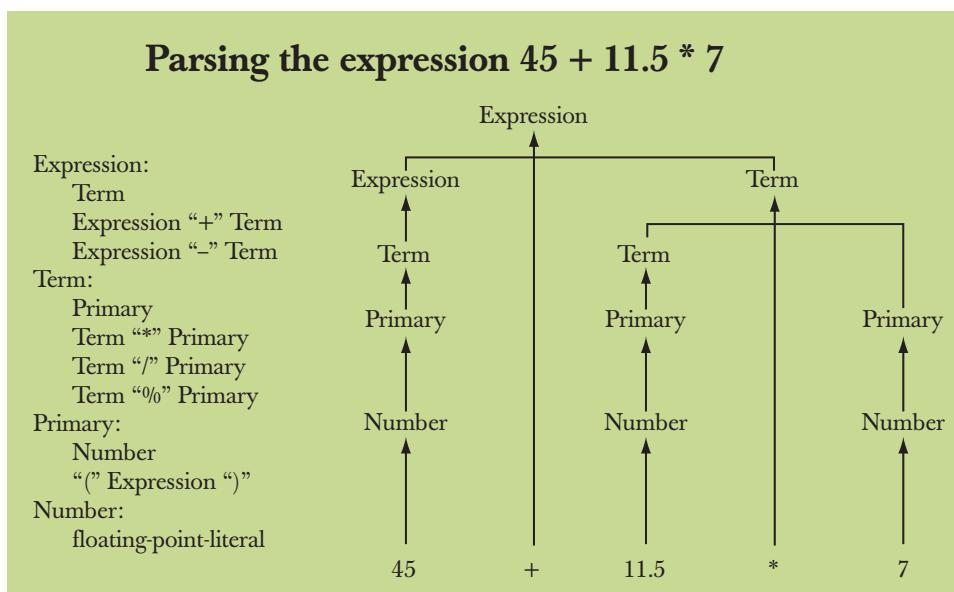
This represents the path we followed through the definitions. Retracing our path, we can say that **2+3** is an **Expression** because **2** is a term which is an **Expression**, **3** is a **Term**, and an **Expression** followed by **+** followed by a **Term** is an **Expression**.

The real reason we are interested in grammars is that they can solve our problem of how to correctly parse expressions with both **+** and *****, so let's try

45+11.5*7. However, “playing computer” following the rules in detail as we did above is tedious, so let’s skip some of the intermediate steps that we have already gone through for **2** and **2+3**. Obviously, **45**, **11.5**, and **7** are all **floating-point-literals** which are **Numbers**, which are **Primary**s, so we can ignore all rules below **Primary**. So we get:

1. **45** is an **Expression** followed by a **+**, so we look for a **Term** to finish the **Expression+Term** rule.
2. **11.5** is a **Term** followed by *****, so we look for a **Primary** to finish the **Term*Primary** rule.
3. **7** is **Primary**, so **11.5*7** is a **Term** according to the **Term*Primary** rule. Now we can see that **45+11.5*7** is an **Expression** according to the **Expression+Term** rule. In particular, it is an **Expression** that first does the multiplication **11.5*7** and then the addition **45+11.5*7**, just as if we had written **45+(11.5*7)**.

Again, we can illustrate this reasoning graphically (again leaving out the **floating-point-literal** to **Number** rule to simplify):



Again, this represents the path we followed through the definitions. Note how the **Term*Primary** rule ensures that **11.5** is multiplied by **7** rather than added to **45**.

You may find this logic hard to follow at first, but many humans do read grammars, and simple grammars are not hard to understand. However, we were not really trying to teach *you* to understand **2+2** or **45+11.5*7**. Obviously, you knew that already. We were trying to find a way for the computer to “understand” **45+11.5*7** and all the other complicated expressions you might give it to evaluate. Actually, complicated grammars are not fit for humans to read, but computers are good at it. They follow such grammar rules quickly and correctly with the greatest of ease. Following precise rules is exactly what computers are good at.

6.4.1 A detour: English grammar

If you have never before worked with grammars, we expect that your head is now spinning. In fact, it may be spinning even if you have seen a grammar before, but take a look at the following grammar for a very small subset of English:

Sentence:

Noun Verb

// e.g., C++ rules

Sentence Conjunction Sentence

// e.g., Birds fly but fish swim

Conjunction:

"and"

"or"

"but"

Noun:

"birds"

"fish"

"C++"

Verb:

"rules"

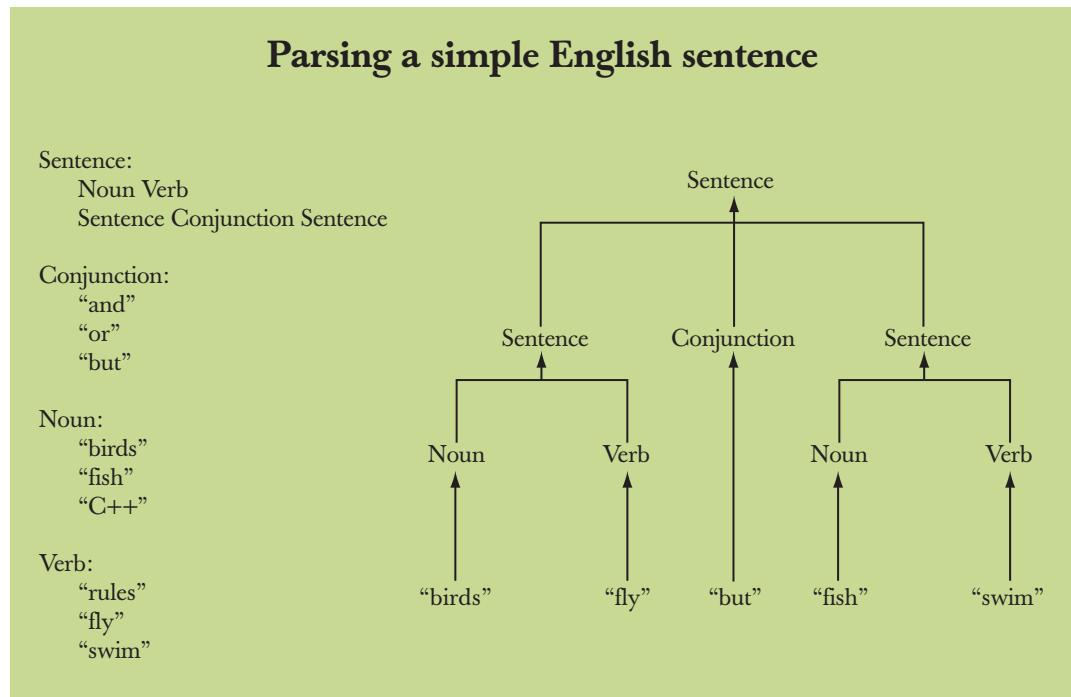
"fly"

"swim"

A sentence is built from parts of speech (e.g., nouns, verbs, and conjunctions). A sentence can be parsed according to these rules to determine which words are nouns, verbs, etc. This simple grammar also includes semantically meaningless sentences such as “C++ fly and birds rules,” but fixing that is a different matter belonging in a far more advanced book.

Many have been taught/shown such rules in middle school or in foreign language class (e.g., English classes). These grammar rules are very fundamental. In fact, there are serious neurological arguments for such rules being hardwired into our brains!

Now look at a parsing tree as we used above for expressions, but used here for simple English:



This is not all that complicated. If you had trouble with §6.4, then please go back and reread it from the beginning; it may make more sense the second time through!

6.4.2 Writing a grammar

How did we pick those expression grammar rules? “Experience” is the honest answer. The way we do it is simply the way people usually write expression grammars. However, writing a simple grammar is pretty straightforward: we need to know how to

- 1. Distinguish a rule from a token
- 2. Put one rule after another (*sequencing*)
- 3. Express alternative patterns (*alternation*)
- 4. Express a repeating pattern (*repetition*)
- 5. Recognize the grammar rule to start with

Different textbooks and different parser systems use different notational conventions and different terminology. For example, some call tokens *terminals* and rules *non-terminals* or *productions*. We simply put tokens in (double) quotes and start with the first rule. Alternatives are put on separate lines. For example:

```
List:
  "{' Sequence "}"
Sequence:
  Element
  Element ", " Sequence
Element:
  "A"
  "B"
```

So a **Sequence** is either an **Element** or an **Element** followed by a **Sequence** using a comma for separation. An **Element** is either the letter **A** or the letter **B**. A **List** is a **Sequence** in “curly brackets.” We can generate these **Lists** (how?):

```
{ A }
{ B }
{ A,B }
{A,A,A,A,B }
```

However, these are not **Lists** (why not?):

```
{ }
A
{ A,A,A,A,B
{A,A,C,A,B }
{ A B C }
{A,A,A,A,B, }
```

This sequence rule is not one you learned in kindergarten or have hardwired into your brain, but it is still not rocket science. See §7.4 and §7.8.1 for examples of how we work with a grammar to express syntactic ideas.

6.5 Turning a grammar into code

There are many ways of getting a computer to follow a grammar. We’ll use the simplest one: we simply write one function for each grammar rule and use our type **Token** to represent tokens. A program that implements a grammar is often called a *parser*.

6.5.1 Implementing grammar rules

To implement our calculator, we need four functions: one to read tokens plus one for each rule in our grammar:

```
get_token()    // read characters and compose tokens
                // uses cin
expression()  // deal with + and -
                // calls term() and get_token()
term()         // deal with *, /, and %
                // calls primary() and get_token()
primary()     // deal with numbers and parentheses
                // calls expression() and get_token()
```

 Note: Each function deals with a specific part of an expression and leaves everything else to other functions; this radically simplifies each function. This is much like a group of humans dealing with problems by letting each person handle problems in his or her own specialty, handing all other problems over to colleagues.

What should these functions actually do? Each function should call other grammar functions according to the grammar rule it is implementing and **get_token()** where a token is required in a rule. For example, when **primary()** tries to follow the **(Expression)** rule, it must call

```
get_token()    // to deal with ( and )
expression()  // to deal with Expression
```

What should such parsing functions return? How about the answer we really wanted? For example, for **2+3**, **expression()** could return **5**. After all, the information is all there. That's what we'll try! Doing so will save us from answering one of the hardest questions from our list: "How do I represent **45+5/7** as data so that I can evaluate it?" Instead of storing a representation of **45+5/7** in memory, we simply evaluate it as we read it from input. This little idea is really a major breakthrough! It will keep the program at a quarter of the size it would have been had we had **expression()** return something complicated for later evaluation. We just saved ourselves about 80% of the work.

The "odd man out" is **get_token()**: because it deals with tokens, not expressions, it can't return the value of a sub-expression. For example, **+** and **(** are not expressions. So, it must return a **Token**. We conclude that we want

```
// functions to match the grammar rules:
Token get_token()    // read characters and compose tokens
double expression() // deal with + and -
```

```
double term()      // deal with *, /, and %
double primary()   // deal with numbers and parentheses
```

6.5.2 Expressions

Let's first write **expression()**. The grammar looks like this:

```
Expression:
  Term
  Expression '+' Term
  Expression '-' Term
```

Since this is our first attempt to turn a set of grammar rules into code, we'll proceed through a couple of false starts. That's the way it usually goes with new techniques, and we learn useful things along the way. In particular, a novice programmer can learn a lot from looking at the dramatically different behavior of similar pieces of code. Reading code is a useful skill to cultivate.

6.5.2.1 Expressions: first try

Looking at the **Expression '+' Term** rule, we try first calling **expression()**, then looking for **+** (and **-**) and then **term()**:

```
double expression()
{
    double left = expression();           // read and evaluate an Expression
    Token t = get_token();                 // get the next token
    switch (t.kind) {                     // see which kind of token it is
        case '+':
            return left + term();           // read and evaluate a Term,
                                              // then do an add
        case '-':
            return left - term();           // read and evaluate a Term,
                                              // then do a subtraction
        default:
            return left;                  // return the value of the Expression
    }
}
```

It looks good. It is almost a trivial transcription of the grammar. It is quite simple, really: first read an **Expression** and then see if it is followed by a **+** or a **-**, and if it is, read the **Term**.

Unfortunately, that doesn't really make sense. How do we know where the expression ends so that we can look for a `+` or a `-`? Remember, our program reads left to right and can't peek ahead to see if a `+` is coming. In fact, this `expression()` will never get beyond its first line: `expression()` starts by calling `expression()` which starts by calling `expression()` and so on "forever." This is called an *infinite recursion* and will in fact terminate after a short while when the computer runs out of memory to hold the "never-ending" sequence of calls of `expression()`. The term *recursion* is used to describe what happens when a function calls itself. Not all recursions are infinite, and recursion is a very useful programming technique (see §8.5.8).

6.5.2.2 Expressions: second try

So what do we do? Every `Term` is an `Expression`, but not every `Expression` is a `Term`; that is, we could start looking for a `Term` and look for a full `Expression` only if we found a `+` or a `-`. For example:

```
double expression()
{
    double left = term();           // read and evaluate a Term
    Token t = get_token();          // get the next token
    switch (t.kind) {               // see which kind of token that is
        case '+':
            return left + expression(); // read and evaluate an Expression,
                                         // then do an add
        case '-':
            return left - expression(); // read and evaluate an Expression,
                                         // then do a subtraction
        default:
            return left;             // return the value of the Term
    }
}
```

This actually – more or less – works. We have tried it in the finished program and it parses every correct expression we throw at it (and no illegal ones). It even correctly evaluates most expressions. For example, `1+2` is read as a `Term` (with the value `1`) followed by `+` followed by an `Expression` (which happens to be a `Term` with the value `2`) and gives the answer `3`. Similarly, `1+2+3` gives `6`. We could go on for quite a long time about what works, but to make a long story short: How about `1-2-3`? This `expression()` will read the `1` as a `Term`, then proceed to read `2-3` as an `Expression` (consisting of the `Term 2` followed by the `Expression 3`). It will then subtract the value of `2-3` from `1`. In other words, it will evaluate `1-(2-3)`. The value of `1-(2-3)` is `2` (positive two). However, we were taught (in primary school or even earlier) that `1-2-3` means `(1-2)-3` and therefore has the value `-4` (negative four).



So we got a very nice program that just didn't do the right thing. That's dangerous. It is especially dangerous because it gives the right answer in many cases. For example, **1+2+3** gives the right answer **(6)** because **1+(2+3)** equals **(1+2)+3**. What fundamentally, from a programming point of view, did we do wrong? We should always ask ourselves this question when we have found an error. That way we might avoid making the same mistake again, and again, and again.

Fundamentally, we just looked at the code and guessed. That's rarely good enough! We have to understand what our code is doing and we have to be able to explain why it does the right thing.

Analyzing our errors is often also the best way to find a correct solution. What we did here was to define **expression()** to first look for a **Term** and then, if that **Term** is followed by a **+** or a **-**, look for an **Expression**. This really implements a slightly different grammar:

```
Expression:
  Term
  Term '+' Expression      // addition
  Term '-' Expression      // subtraction
```

The difference from our desired grammar is exactly that we wanted **1-2-3** to be the **Expression 1-2** followed by **-** followed by the **Term 3**, but what we got here was the **Term 1** followed by **-** followed by the **Expression 2-3**; that is, we wanted **1-2-3** to mean **(1-2)-3** but we got **1-(2-3)**.

Yes, debugging can be tedious, tricky, and time-consuming, but in this case we are really working through rules you learned in primary school and learned to apply without too much trouble. The snag is that we have to teach the rules to a computer – and a computer is a far slower learner than you are.

Note that we could have defined **1-2-3** to mean **1-(2-3)** rather than **(1-2)-3** and avoided this discussion altogether. Often, the trickiest programming problems come when we must match conventional rules that were established by and for humans long before we started using computers.

6.5.2.3 Expressions: third time lucky

So, what now? Look again at the grammar (the correct grammar in §6.5.2): any **Expression** starts with a **Term** and such a **Term** can be followed by a **+** or a **-**. So, we have to look for a **Term**, see if it is followed by a **+** or a **-**, and keep doing that until there are no more plusses or minuses. For example:

```
double expression()
{
  double left = term();          // read and evaluate a Term
  Token t = get_token();          // get the next token
```

```

while (t.kind=='+' || t.kind=='-') {    // look for a + or a -
    if (t.kind == '+')
        left += term();                // evaluate Term and add
    else
        left -= term();                // evaluate Term and subtract
    t = get_token();
}
return left;                         // finally: no more + or -; return the answer
}

```

This is a bit messier: we had to introduce a loop to keep looking for plusses and minuses. We also got a bit repetitive: we test for `+` and `-` twice and twice call `get_token()`. Because it obscures the logic of the code, let's just get rid of the duplication of the test for `+` and `-`:

```

double expression()
{
    double left = term();           // read and evaluate a Term
    Token t = get_token();          // get the next token
    while (true) {
        switch (t.kind) {
            case '+':
                left += term();      // evaluate Term and add
                t = get_token();
                break;
            case '-':
                left -= term();      // evaluate Term and subtract
                t = get_token();
                break;
            default:
                return left;         // finally: no more + or -; return the answer
        }
    }
}

```

Note that – except for the loop – this is actually rather similar to our first try (§6.5.2.1). What we have done is to remove the mention of `expression()` within `expression()` and replace it with a loop. In other words, we translated the **Expression** in the grammar rules for **Expression** into a loop looking for a **Term** followed by a `+` or a `-`.

6.5.3 Terms

The grammar rule for **Term** is very similar to the **Expression** rule:

Term:

Primary
Term '*' Primary
Term '/' Primary
Term '%' Primary

Consequently, the code should be very similar also. Here is a first try:

```
double term()
{
    double left = primary();
    Token t = get_token();
    while (true) {
        switch (t.kind) {
            case '*':
                left *= primary();
                t = get_token();
                break;
            case '/':
                left /= primary();
                t = get_token();
                break;
            case '%':
                left %= primary();
                t = get_token();
                break;
            default:
                return left;
        }
    }
}
```

Unfortunately, this doesn't compile: the remainder operation (%) is not defined for floating-point numbers. The compiler kindly tells us so. When we answered question 5 in §6.3.5 – "Should we also allow floating-point inputs?" – with a confident "Sure!" we actually hadn't thought the issue through and fell victim to *feature creep*. That *always* happens! So what do we do about it? We could at run time check that both operands of % are integers and give an error if they are not. Or we could simply leave % out of our calculator. Let's take the simplest choice for now. We can always add % later; see §7.5.

After we eliminate the % case, the function works: terms are correctly parsed and evaluated. However, an experienced programmer will notice an undesirable detail that makes **term()** unacceptable. What would happen if you entered **2/0?**



You can't divide by zero. If you try, the computer hardware will detect it and terminate your program with a somewhat unhelpful error message. An inexperienced programmer will discover this the hard way. So, we'd better check and give a decent error message:

```
double term()
{
    double left = primary();
    Token t = get_token();
    while (true) {
        switch (t.kind) {
            case '*' :
                left *= primary();
                t = get_token();
                break;
            case '/' :
                { double d = primary();
                    if (d == 0) error("divide by zero");
                    left /= d;
                    t = get_token();
                    break;
                }
            default:
                return left;
        }
    }
}
```

Why did we put the statements handling `/` into a block? The compiler insists. If you want to define and initialize variables within a `switch`-statement, you must place them inside a block.

6.5.4 Primary expressions

The grammar rule for primary expressions is also simple:

```
Primary:
Number
(' Expression ')
```

The code that implements it is a bit messy because there are more opportunities for syntax errors:

```

double primary()
{
    Token t = get_token();
    switch (t.kind) {
        case '(': // handle '(' expression ')'
            {
                double d = expression();
                t = get_token();
                if (t.kind != ')') error("')' expected");
                return d;
            }
        case '8': // we use '8' to represent a number
            return t.value; // return the number's value
        default:
            error("primary expected");
    }
}

```

Basically there is nothing new compared to `expression()` and `term()`. We use the same language primitives, the same way of dealing with `Tokens`, and the same programming techniques.

6.6 Trying the first version

To run these calculator functions, we need to implement `get_token()` and provide a `main()`. The `main()` is trivial: we just keep calling `expression()` and printing out its result:

```

int main()
{
    try {
        while (cin)
            cout << expression() << '\n';
            keep_window_open();
    }
    catch (exception& e) {
        cerr << e.what() << '\n';
        keep_window_open ();
        return 1;
    }
    catch (...) {
        cerr << "exception \n";
        keep_window_open ();
        return 2;
    }
}

```

The error handling is the usual “boilerplate” (§5.6.3). Let us postpone the description of the implementation of `get_token()` to §6.8 and test this first version of the calculator.

TRY THIS



This first version of the calculator program (including `get_token()`) is available as file `calculator00.cpp`. Get it to run and try it out.

Unsurprisingly, this first version of the calculator doesn’t work quite as we expected. So we shrug and ask, “Why not?” or rather, “So, why does it work the way it does?” and “What does it do?” Type a **2** followed by a newline. No response. Try another newline to see if it’s asleep. Still no response. Type a **3** followed by a newline. No response! Type a **4** followed by a newline. It answers **2!** Now the screen looks like this:

2
3
4
2

We carry on by typing **5+6**. The program responds with a **5**, so that the screen looks like this:

2
3
4
2
5+6
5

Unless you have programmed before, you are most likely very puzzled! In fact, even an experienced programmer might be puzzled. What’s going on here? At this point, you try to get out of the program. How do you do this? We “forgot” to program an exit command, but an error will cause the program to exit, so you type an **x** and the program prints **Bad token** and exits. Finally, something worked as planned!

However, we forgot to distinguish between input and output on the screen. Before we try to solve the main puzzle, let’s just fix the output to better see what we are doing. Adding an **=** to indicate output will do for now:

```
while (cin) cout << "=" << expression() << '\n'; // version 1
```

Now, entering the exact sequence of characters as before, we get

```
2  
3  
4  
= 2  
5+6  
= 5  
x  
Bad token
```

Strange! Try to figure out what the program did. We tried another few examples, but let's just look at this. This is a puzzle:

Why didn't the program respond after the first **2** and **3** and the newlines?

Why did the program respond with **2**, rather than **4**, after we entered **4**?

Why did the program answer **5**, rather than **11**, after **5+6**?

There are many possible ways of proceeding from such mysterious results. We'll examine some of those in the next chapter, but here, let's just think. Could the program be doing bad arithmetic? That's most unlikely; the value of **4** isn't **2**, and the value of **5+6** is **11** rather than **5**. Consider what happens when we enter **1 2 3 4+5 6+7 8+9 10 11 12** followed by a newline. We get

```
1 2 3 4+5 6+7 8+9 10 11 12  
= 1  
= 4  
= 6  
= 8  
= 10
```

Huh? No **2** or **3**. Why **4** and not **9** (that is, **4+5**)? Why **6** and not **13** (that is, **6+7**)? Look carefully: the program is outputting every third token! Maybe the program “eats” some of our input without evaluating it? It does. Consider **expression()**:

```
double expression()  
{  
    double left = term();           // read and evaluate a Term  
    Token t = get_token();          // get the next token  
    while (true) {  
        switch (t.kind) {
```

```
case '+':
    left += term();    // evaluate Term and add
    t = get_token();
    break;
case '-':
    left -= term();    // evaluate Term and subtract
    t = get_token();
    break;
default:
    return left;      // finally: no more + or -; return the answer
}
}
```

When the `Token` returned by `get_token()` is not a `+` or a `-` we just return. We don't use that token and we don't store it anywhere for any other function to use later. That's not smart. Throwing away input without even determining what it is can't be a good idea. A quick look shows that `term()` has exactly the same problem. That explains why our calculator ate two tokens for each that it used.

Let us modify **expression()** so that it doesn't "eat" tokens. Where would we put that next token (**t**) when the program doesn't need it? We could think of many elaborate schemes, but let's jump to the obvious answer ("obvious" once you see it): that token is going to be used by some other function that is reading tokens from the input, so let's put the token back into the input stream so that it can be read again by some other function! Actually, you can put characters back into an **istream**, but that's not really what we want. We want to deal with tokens, not mess with characters. What we want is an input stream that deals with tokens and that you can put an already read token back into.

So, assume that we have a stream of tokens – a “**Token_stream**” – called **ts**. Assume further that a **Token_stream** has a member function **get()** that returns the next token and a member function **putback(t)** that puts a token **t** back into the stream. We’ll implement that **Token_stream** in §6.8 as soon as we have had a look at how it needs to be used. Given **Token_stream**, we can rewrite **expression()** so that it puts a token that it does not use back into the **Token_stream**:

```
double expression()
{
    double left = term();           // read and evaluate a Term
    Token t = ts.get();            // get the next Token from the Token stream

    while (true) {
        switch (t.kind) {
```

```

case '+':
    left += term(); // evaluate Term and add
    t = ts.get();
    break;
case '-':
    left -= term(); // evaluate Term and subtract
    t = ts.get();
    break;
default:
    ts.putback(t); // put t back into the token stream
    return left; // finally: no more + or -; return the answer
}
}
}

```

In addition, we must make the same change to **term()**:

```

double term()
{
    double left = primary();
    Token t = ts.get(); // get the next Token from the Token stream

    while (true) {
        switch (t.kind) {
            case '*':
                left *= primary();
                t = ts.get();
                break;
            case '/':
            {
                double d = primary();
                if (d == 0) error("divide by zero");
                left /= d;
                t = ts.get();
                break;
            }
            default:
                ts.putback(t); // put t back into the Token stream
                return left;
        }
    }
}

```

For our last parser function, `primary()`, we just need to change `get_token()` to `ts.get()`; `primary()` uses every token it reads.

6.7 Trying the second version

So, we are ready to test our second version. This second version of the calculator program (including `Token_stream`) is available as file `calculator01.cpp`. Get it to run and try it out. Type **2** followed by a newline. No response. Try another newline to see if it's asleep. Still no response. Type a **3** followed by a newline and it answers **2**. Try **2+2** followed by a newline and it answers **3**. Now your screen looks like this:

```
2  
3  
=2  
2+2  
=3
```

Hmm. Maybe our introduction of `putback()` and its use in `expression()` and `term()` didn't fix the problem. Let's try another test:

```
2 3 4 2+3 2*3  
= 2  
= 3  
= 4  
= 5
```

Yes! These are correct answers! But the last answer (**6**) is missing. We still have a token-look-ahead problem. However, this time the problem is not that our code “eats” characters, but that it doesn’t get any output for an expression until we enter the following expression. The result of an expression isn’t printed immediately; the output is postponed until the program has seen the first token of the next expression. Unfortunately, the program doesn’t see that token until we hit Return after the next expression. The program isn’t really wrong; it is just a bit slow responding.

How can we fix this? One obvious solution is to require a “print command.” So, let’s accept a semicolon after an expression to terminate it and trigger output. And while we are at it, let’s add an “exit command” to allow for graceful exit. The character **q** (for “quit”) would do nicely for an exit command. In `main()`, we have

```
while (cin) cout << "=" << expression() << '\n'; // version 1
```

We can change that to the messier but more useful

```
double val = 0;
while (cin) {
    Token t = ts.get();

    if (t.kind == 'q') break; // 'q' for "quit"
    if (t.kind == ';') // ';' for "print now"
        cout << "=" << val << '\n';
    else
        ts.putback(t);
    val = expression();
}
```

Now the calculator is actually usable. For example, we get

```
2;
= 2
2+3;
= 5
3+4*5;
= 23
q
```

At this point we have a good initial version of the calculator. It's not quite what we really wanted, but we have a program that we can use as the base for making a more acceptable version. Importantly, we can now correct problems and add features one by one while maintaining a working program as we go along.

6.8 Token streams

Before further improving our calculator, let us show the implementation of **Token_stream**. After all, nothing – nothing at all – works until we get correct input. We implemented **Token_stream** first of all but didn't want too much of a digression from the problems of calculation before we had shown a minimal solution.

Input for our calculator is a sequence of tokens, just as we showed for **(1.5+4)*11** above (§6.3.3). What we need is something that reads characters from the standard input, **cin**, and presents the program with the next token when it asks for it. In addition, we saw that we – that is, our calculator program – often read a token too many, so that we must be able to put it back for later use. This is typical and fundamental; when you see **1.5+4** reading strictly left to right, how could you know that the number **1.5** had been completely read without reading

the `+`? Until we see the `+` we might be on our way to reading **1.55555**. So, we need a “stream” that produces a token when we ask for one using `get()` and where we can put a token back into the stream using `putback()`. Everything we use in C++ has a type, so we have to start by defining the type **Token_stream**.

You probably noticed the `public:` in the definition of **Token** in §6.3.3. There, it had no apparent purpose. For **Token_stream**, we need it and must explain its function. A C++ user-defined type often consists of two parts: the public interface (labeled `public:`) and the implementation details (labeled `private:`). The idea is to separate what a user of a type needs for convenient use from the details that we need in order to implement the type, but that we’d rather not have users mess with:

```
class Token_stream {
public:
    // user interface
private:
    // implementation details
    // (not directly accessible to users of Token_stream)
};
```

Obviously, users and implementers are often just us “playing different roles,” but making the distinction between the (public) interface meant for users and the (private) implementation details used only by the implementer is a powerful tool for structuring code. The public interface should contain (only) what a user needs, which is typically a set of functions. The private implementation contains what is necessary to implement those public functions, typically data and functions dealing with messy details that the users need not know about and shouldn’t directly use.

Let’s elaborate the **Token_stream** type a bit. What does a user want from it? Obviously, we want `get()` and `putback()` functions – that’s why we invented the notion of a token stream. The **Token_stream** is to make **Tokens** out of characters that it reads for input, so we need to be able to make a **Token_stream** and to define it to read from `cin`. Thus, the simplest **Token_stream** looks like this:

```
class Token_stream {
public:
    Token_stream();           // make a Token_stream that reads from cin
    Token get();              // get a Token
    void putback(Token t);    // put a Token back
private:
    // implementation details
};
```

That’s all a user needs to use a **Token_stream**. Experienced programmers will wonder why `cin` is the only possible source of characters, but we decided to take our input from the keyboard. We’ll revisit that decision in a Chapter 7 exercise.

Why do we use the “verbose” name **putback()** rather than the logically sufficient **put()**? We wanted to emphasize the asymmetry between **get()** and **putback()**; this is an input stream, not something that you can also use for general output. Also, **istream** has a **putback()** function: consistency in naming is a useful property of a system. It helps people remember and helps people avoid errors.

We can now make a **Token_stream** and use it:

```
Token_stream ts;           // a Token_stream called ts
Token t = ts.get();       // get next Token from ts
// ...
ts.putback(t);            // put the Token t back into ts
```

That’s all we need to write the rest of the calculator.

6.8.1 Implementing **Token_stream**

Now, we need to implement those three **Token_stream** functions. How do we represent a **Token_stream**? That is, what data do we need to store in a **Token_stream** for it to do its job? We need space for any token we put back into the **Token_stream**. To simplify, let’s say we can put back at most one token at a time. That happens to be sufficient for our program (and for many, many similar programs). That way, we just need space for one **Token** and an indicator of whether that space is full or empty:

```
class Token_stream {
public:
    Token get();           // get a Token (get() is defined in §6.8.2)
    void putback(Token t); // put a Token back
private:
    bool full {false};     // is there a Token in the buffer?
    Token buffer; // here is where we keep a Token put back using putback()
};
```

Now we can define (“write”) the two member functions. The **putback()** is easy, so we will define it first. The **putback()** member function puts its argument back into the **Token_stream**’s buffer:

```
void Token_stream::putback(Token t)
{
    buffer = t;           // copy t to buffer
    full = true;          // buffer is now full
}
```

The keyword **void** (meaning “nothing”) is used to indicate that **putback()** doesn’t return a value.

When we define a member of a class outside the class definition itself, we have to mention which class we mean the member to be a member of. We use the notation

```
class_name :: member_name
```

for that. In this case, we define **Token_stream**’s member **putback**.

Why would we define a member outside its class? The main answer is clarity: the class definition (primarily) states what the class can do. Member function definitions are implementations that specify how things are done. We prefer to put them “elsewhere” where they don’t distract. Our ideal is to have every logical entity in a program fit on a screen. Class definitions typically do that if the member function definitions are placed elsewhere, but not if they are placed within the class definition (“in-class”).

If we wanted to make sure that we didn’t try to use **putback()** twice without reading what we put back in between (using **get()**), we could add a test:

```
void Token_stream::putback(Token t)
{
    if (full) error("putback() into a full buffer");
    buffer = t;           // copy t to buffer
    full = true;          // buffer is now full
}
```

The test of **full** checks the pre-condition “There is no **Token** in the buffer.”

Obviously, a **Token_stream** should start out empty. That is, **full** should be **false** until after the first call of **get()**. We achieve that by initializing the member **full** right in the definition of **Token_stream**.

6.8.2 Reading tokens

All the real work is done by **get()**. If there isn’t already a **Token** in **Token_stream::buffer**, **get()** must read characters from **cin** and compose them into **Tokens**:

```
Token Token_stream::get()
{
    if (full) {                      // do we already have a Token ready?
        full = false;                // remove Token from buffer
        return buffer;
    }
```

```

char ch;
cin >> ch; // note that >> skips whitespace (space, newline, tab, etc.)

switch (ch) {
    case ';' :           // for "print"
    case 'q' :           // for "quit"
    case '(' : case ')' : case '+' : case '-' : case '*' : case '/' :
        return Token{ch};           // let each character represent itself
    case '!' :
    case '0' : case '1' : case '2' : case '3' : case '4' :
    case '5' : case '6' : case '7' : case '8' : case '9' :
        { cin.putback(ch);           // put digit back into the input stream
            double val;
            cin >> val;             // read a floating-point number
            return Token{'8',val};     // let '8' represent "a number"
        }
    default:
        error("Bad token");
}
}

```

Let's examine `get()` in detail. First we check if we already have a `Token` in the buffer. If so, we can just return that:

```

if (full) {           // do we already have a Token ready?
    full = false;       // remove Token from buffer
    return buffer;
}

```

Only if `full` is `false` (that is, there is no token in the buffer) do we need to mess with characters. In that case, we read a character and deal with it appropriately. We look for parentheses, operators, and numbers. Any other character gets us the call of `error()` that terminates the program:

```

default:
    error("Bad token");

```

The `error()` function is described in §5.6.3 and we make it available in `std_lib_facilities.h`.

We had to decide how to represent the different kinds of `Tokens`; that is, we had to choose values for the member `kind`. For simplicity and ease of debugging,

we decided to let the **kind** of a **Token** be the parentheses and operators themselves. This leads to extremely simple processing of parentheses and operators:

```
case '(': case ')': case '+': case '-': case '*': case '/':
    return Token{ch};      // let each character represent itself
```

To be honest, we had forgotten **';'** for “print” and **'q'** for “quit” in our first version. We didn’t add them until we needed them for our second solution.

6.8.3 Reading numbers

Now we just have to deal with numbers. That’s actually not that easy. How do we really find the value of **123**? Well, that’s **100+20+3**, but how about **12.34**, and should we accept scientific notation, such as **12.34e5**? We could spend hours or days to get this right, but fortunately, we don’t have to. Input streams know what C++ literals look like and how to turn them into values of type **double**. All we have to do is to figure out how to tell **cin** to do that for us inside **get()**:

```
case '.':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
{   cin.putback(ch);           // put digit back into the input stream
    double val;
    cin >> val;             // read a floating-point number
    return Token{'8',val};     // let '8' represent "a number"
}
```

We – somewhat arbitrarily – chose **'8'** to represent “a number” in a **Token**.

How do we know that a number is coming? Well, if we guess from experience or look in a C++ reference (e.g., Appendix A), we find that a numeric literal must start with a digit or . (the decimal point). So, we test for that. Next, we want to let **cin** read the number, but we have already read the first character (a digit or dot), so just letting **cin** loose on the rest will give a wrong result. We could try to combine the value of the first character with the value of “the rest” as read by **cin**; for example, if someone typed **123**, we would get **1** and **cin** would read **23** and we’d have to add **100** to **23**. Yuck! And that’s a trivial case. Fortunately (and not by accident), **cin** works much like **Token_stream** in that you can put a character back into it. So instead of doing any messy arithmetic, we just put the initial character back into **cin** and then let **cin** read the whole number.

Please note how we again and again avoid doing complicated work and instead find simpler solutions – often relying on library facilities. That's the essence of programming: the continuing search for simplicity. Sometimes that's – somewhat facetiously – expressed as “Good programmers are lazy.” In that sense (and only in that sense), we should be “lazy”; why write a lot of code if we can find a way of writing far less?



6.9 Program structure

Sometimes, the proverb says, it's hard to see the forest for the trees. Similarly, it is easy to lose sight of a program when looking at all its functions, classes, etc. So, let's have a look at the program with its details omitted:

```
#include "std_lib_facilities.h"

class Token {/* ... */;
class Token_stream {/* ... */;

void Token_stream::putback(Token t) {/* ... */
Token Token_stream::get() {/* ... */

Token_stream ts;           // provides get() and putback()
double expression()        // declaration so that primary() can call expression()

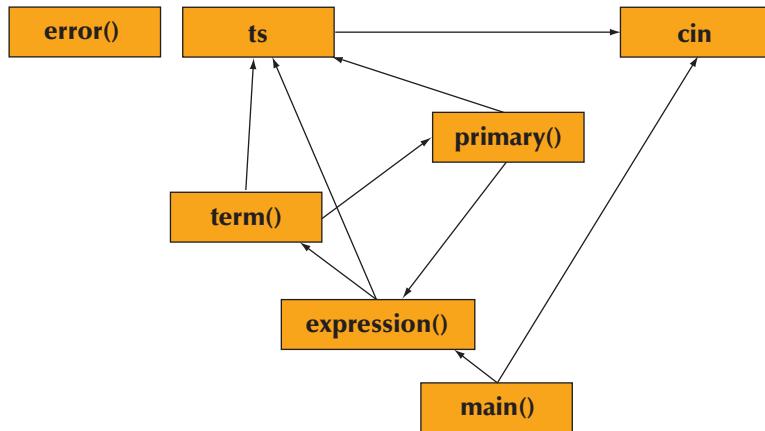
double primary() {/* ... */}    // deal with numbers and parentheses
double term() {/* ... */}      // deal with * and /
double expression() {/* ... */} // deal with + and -

int main() {/* ... */}        // main loop and deal with errors
```



The order of the declarations is important. You cannot use a name before it has been declared, so **ts** must be declared before **ts.get()** uses it, and **error()** must be declared before the parser functions because they all use it. There is an interesting loop in the call graph: **expression()** calls **term()** which calls **primary()** which calls **expression()**.

We can represent that graphically (leaving out calls to `error()` – everyone calls `error()`):



This means that we can't just define those three functions: there is no order that allows us to define every function before it is used. We need at least one declaration that isn't also a definition. We chose to declare ("forward declare") `expression()`.

But does this work? It does, for some definition of "work." It compiles, runs, correctly evaluates expressions, and gives decent error messages. But does it work in a way that we like? The unsurprising answer is "Not really." We tried the first version in §6.6 and removed a serious bug. This second version (§6.7) is not much better. But that's fine (and expected). It is good enough for its main purpose, which is to be something that we can use to verify our basic ideas and get feedback from. As such, it is a success, but try it: it'll (still) drive you nuts!

TRY THIS



Get the calculator as presented above to run, see what it does, and try to figure out why it works as it does.



Drill

This drill involves a series of modifications of a buggy program to turn it from something useless into something reasonably useful.

1. Take the calculator from the file `calculator02buggy.cpp`. Get it to compile. You need to find and fix a few bugs. Those bugs are not in the text in the book. Find the three logic errors deviously inserted in `calculator02buggy.cpp` and remove them so that the calculator produces correct results.
2. Change the character used as the exit command from `q` to `x`.
3. Change the character used as the print command from `;` to `=`.
4. Add a greeting line in `main()`:

`"Welcome to our simple calculator.
Please enter expressions using floating-point numbers."`

5. Improve that greeting by mentioning which operators are available and how to print and exit.

Review

1. What do we mean by “Programming is understanding”?
2. The chapter details the creation of a calculator program. Write a short analysis of what the calculator should be able to do.
3. How do you break a problem up into smaller manageable parts?
4. Why is creating a small, limited version of a program a good idea?
5. Why is feature creep a bad idea?
6. What are the three main phases of software development?
7. What is a “use case”?
8. What is the purpose of testing?
9. According to the outline in the chapter, describe the difference between a **Term**, an **Expression**, a **Number**, and a **Primary**.
10. In the chapter, an input was broken down into its component **Terms**, **Expressions**, **Primarys**, and **Numbers**. Do this for `(17+4)/(5-1)`.
11. Why does the program not have a function called `number()`?
12. What is a token?
13. What is a grammar? A grammar rule?
14. What is a class? What do we use classes for?
15. How can we provide a default value for a member of a class?
16. In the expression function, why is the default for the **switch**-statement to “put back” the token?

17. What is “look-ahead”?
18. What does `putback()` do and why is it useful?
19. Why is the remainder (modulus) operation, `%`, difficult to implement in the `term()`?
20. What do we use the two data members of the `Token` class for?
21. Why do we (sometimes) split a class’s members into `private` and `public` members?
22. What happens in the `Token_stream` class when there is a token in the buffer and the `get()` function is called?
23. Why were the `'.'` and `'q'` characters added to the `switch`-statement in the `get()` function of the `Token_stream` class?
24. When should we start testing our program?
25. What is a “user-defined type”? Why would we want one?
26. What is the interface to a C++ “user-defined type”?
27. Why do we want to rely on libraries of code?

Terms

analysis	grammar	prototype
class	implementation	pseudo code
class member	interface	public
data member	member function	syntax analyzer
design	parser	token
divide by zero	private	use case

Exercises

1. If you haven’t already, do the **Try this** exercises from this chapter.
2. Add the ability to use `{}` as well as `()` in the program, so that `{(4+5)*6} / (3+4)` will be a valid expression.
3. Add a factorial operator: use a suffix `!` operator to represent “factorial.” For example, the expression `7!` means `7 * 6 * 5 * 4 * 3 * 2 * 1`. Make `!` bind tighter than `*` and `/`; that is, `7*8!` means `7*(8!)` rather than `(7*8)!`. Begin by modifying the grammar to account for a higher-level operator. To agree with the standard mathematical definition of factorial, let `0!` evaluate to `1`. Hint: The calculator functions deal with `doubles`, but factorial is defined only for `ints`, so just for `x!`, assign the `x` to an `int` and calculate the factorial of that `int`.
4. Define a class `Name_value` that holds a string and a value. Rework exercise 19 in Chapter 4 to use a `vector<Name_value>` instead of two `vectors`.
5. Add the article `the` to the “English” grammar in §6.4.1, so that it can describe sentences such as “The birds fly but the fish swim.”

6. Write a program that checks if a sentence is correct according to the “English” grammar in §6.4.1. Assume that every sentence is terminated by a full stop (.) surrounded by whitespace. For example, **birds fly but the fish swim .** is a sentence, but **birds fly but the fish swim** (terminating dot missing) and **birds fly but the fish swim.** (no space before dot) are not. For each sentence entered, the program should simply respond “OK” or “not OK.” Hint: Don’t bother with tokens; just read into a **string** using `>>`.
7. Write a grammar for bitwise logical expressions. A bitwise logical expression is much like an arithmetic expression except that the operators are **!** (not), **~** (complement), **&** (and), **|** (or), and **^** (exclusive or). Each operator does its operation to each bit of its integer operands (see §25.5). **!** and **~** are prefix unary operators. A **^** binds tighter than a **|** (just as ***** binds tighter than **+**) so that **x|y^z** means **x|(y^z)** rather than **(x|y)^z**. The **&** operator binds tighter than **^** so that **x^y&z** means **x^(y&z)**.
8. Redo the “Bulls and Cows” game from exercise 12 in Chapter 5 to use four letters rather than four digits.
9. Write a program that reads digits and composes them into integers. For example, **123** is read as the characters 1, 2, and 3. The program should output **123 is 1 hundred and 2 tens and 3 ones**. The number should be output as an **int** value. Handle numbers with one, two, three, or four digits. Hint: To get the integer value **5** from the character **'5'** subtract **'0'**, that is, **'5' - '0' == 5**.
10. A permutation is an ordered subset of a set. For example, say you wanted to pick a combination to a vault. There are 60 possible numbers, and you need three different numbers for the combination. There are $P(60,3)$ permutations for the combination, where P is defined by the formula

$$P(a,b) = \frac{a!}{(a-b)!},$$

where **!** is used as a suffix factorial operator. For example, **4!** is **4*3*2*1**.

Combinations are similar to permutations, except that the order of the objects doesn’t matter. For example, if you were making a “banana split” sundae and wished to use three different flavors of ice cream out of five that you had, you wouldn’t care if you used a scoop of vanilla at the beginning or the end; you would still have used vanilla. The formula for combinations is

$$C(a,b) = \frac{P(a,b)}{b!}.$$

Design a program that asks users for two numbers, asks them whether they want to calculate permutations or combinations, and prints out the result. This will have several parts. Do an analysis of the above requirements. Write exactly what the program will have to do. Then, go into the design phase. Write pseudo code for the program, and break it into sub-components. This program should have error checking. Make sure that all erroneous inputs will generate good error messages.

Postscript

Making sense of input is one of the fundamental programming activities. Every program somehow faces that problem. Making sense of something directly produced by a human is among the hardest problems. For example, many aspects of voice recognition are still a research problem. Simple variations of this problem, such as our calculator, cope by using a grammar to define the input.



Completing a Program

“It ain’t over till the fat lady sings.”

—Opera proverb

Writing a program involves gradually refining your ideas of what you want to do and how you want to express it. In Chapter 6, we produced the initial working version of a calculator program. Here, we’ll refine it. Completing the program – that is, making it fit for users and maintainers – involves improving the user interface, doing some serious work on error handling, adding a few useful features, and restructuring the code for ease of understanding and modification.

7.1 Introduction
7.2 Input and output
7.3 Error handling
7.4 Negative numbers
7.5 Remainder: %
7.6 Cleaning up the code
7.6.1 Symbolic constants
7.6.2 Use of functions
7.6.3 Code layout
7.6.4 Commenting

7.7 Recovering from errors
7.8 Variables
7.8.1 Variables and definitions
7.8.2 Introducing names
7.8.3 Predefined names
7.8.4 Are we there yet?

7.1 Introduction

When your program first starts running “reasonably,” you’re probably about halfway finished. For a large program or a program that could do harm if it misbehaved, you will be nowhere near halfway finished. Once the program “basically works,” the real fun begins! That’s when we have enough working code to experiment with ideas.

In this chapter, we will guide you through the considerations a professional programmer might have trying to improve the calculator from Chapter 6. Note that the questions asked about the program and the issues considered here are far more interesting than the calculator itself. What we do is to give an example of how real programs evolve under the pressure of requirements and constraints and of how a programmer can gradually improve code.

7.2 Input and output

If you look back to the beginning of Chapter 6, you’ll find that we decided to prompt the user with

Expression:

and to report back answers with

Result:

In the heat of getting the program to run, we forgot all about that. That’s pretty typical. We can’t think of everything all the time, so when we stop to reflect, we find that we have forgotten something.

For some programming tasks, the initial requirements cannot be changed. That’s usually too rigid a policy and leads to programs that are unnecessarily poor solutions to the problems that they are written to solve. So, let’s consider

what we would do, assuming that we can change the specification of what exactly the program should do. Do we really want the program to write **Expression:** and **Result:**? How would we know? Just “thinking” rarely helps. We have to try and see what works best.

```
2+3; 5*7; 2+9;
```

currently gives

```
= 5  
= 35  
= 11
```

If we used **Expression:** and **Result:**, we'd get

```
Expression: 2+3; 5*7; 2+9;  
Result : 5  
Expression: Result: 35  
Expression: Result: 11  
Expression:
```

We are sure that some people will like one style and others will like the other. In such cases, we can consider giving people a choice, but for this simple calculator that would be overkill, so we must decide. We think that writing **Expression:** and **Result:** is a bit too “heavy” and distracting. Using those, the actual expressions and results are only a minor part of what appears on the screen, and since expressions and results are what matters, nothing should distract from them. On the other hand, unless we somehow separate what the user types from what the computer outputs, the result can be confusing. During initial debugging, we added **=** as a result indicator. We would also like a short “prompt” to indicate that the program wants input. The **>** character is often used as a prompt:

```
> 2+3;  
= 5  
> 5*7;  
= 35  
>
```

This looks much better, and we can get it by a minor change to the main loop of **main()**:

```
double val = 0;  
while (cin) {  
    cout << "> " ;    // print prompt  
    Token t = ts.get();
```

```

if (t.kind == 'q') break;
if (t.kind == ';')
    cout << "=" << val << '\n';      // print result
else
    ts.putback(t);
val = expression();
}

```

Unfortunately, the result of putting several expressions on a line is still messy:

```

> 2+3; 5*7; 2+9;
= 5
> = 35
> = 11
>

```

The basic problem is that we didn't think of multiple expressions on a line when we started out (at least we pretended not to). What we want is

```

> 2+3; 5*7; 2+9;
= 5
= 35
= 11
>

```

This looks right, but unfortunately there is no really obvious way of achieving it. We first looked at `main()`. Is there a way to write out `>` only if it is not immediately followed by a `=`? We cannot know! We need to write `>` before the `get()`, but we do not know if `get()` actually reads new characters or simply gives us a **Token** from characters that it had already read from the keyboard. In other words, we would have to mess with `Token_stream` to make this final improvement.

For now, we decide that what we have is good enough. If we find that we have to modify `Token_stream`, we'll revisit this decision. However, it is unwise to make major structural changes to gain a minor advantage, and we haven't yet thoroughly tested the calculator.

7.3 Error handling



The first thing to do once you have a program that “basically works” is to try to break it; that is, we try to feed it input in the hope of getting it to misbehave. We say “hope” because the challenge here is to find as many errors as possible, so

that you can fix them before anybody else finds them. If you go into this exercise with the attitude that “my program works and I don’t make errors!” you won’t find many bugs and you’ll feel bad when you do find one. You’d be playing head games with yourself! The right attitude when testing is “I’ll break it! I’m smarter than any program – even my own!” So, we feed the calculator a mix of correct and incorrect expressions. For example:

```
1+2+3+4+5+6+7+8
1-2-3-4
!+2
;;
(1+3;
(1+;
1*2/3%4+5-6;
0;
1+;
+1
1++;
1/0
1/0;
1++2;
-2;
-2;;;
1234567890123456;
'a';
q
1+q
1+2; q
```

TRY THIS



Feed a few such “problematic” expressions to the calculator and try to figure out in how many ways you can get it to misbehave. Can you get it to crash, that is, to get it past our error handling and give a machine error? We don’t think you can. Can you get it to exit without a useful error message? You can.

Technically, this is known as *testing*. There are people who do this – break programs – for a living. Testing is a very important part of software development and can actually be fun. Chapter 26 examines testing in some detail. One big question

is: “Can we test the program systematically, so that we find all of the errors?” There is no general answer to this question; that is, there is no answer that holds for all programs. However, you can do rather well for many programs when you approach testing seriously. You try to create test cases systematically, and just in case your strategy for selecting tests isn’t complete, you do some “unreasonable” tests, such as

Mary had a little lamb
srtvrqtiewcbet7rewaevre-wqcncntrretewru754389652743nvcqnwq;
!@#\$%^&*()~:;

Once, when testing compilers, I got into the habit of feeding email that reported compiler errors straight to the compiler – mail headers, user’s explanation, and all. That wasn’t “sensible” because “nobody would do that.” However, a program ideally catches all errors, not just the sensible ones, and soon that compiler was very resilient against “strange input.”

The first really annoying thing we noticed when testing the calculator was that the window closed immediately after inputs such as

**+1;
0
!+2**

A little thought (or some tracing of the program’s execution) shows that the problem is that the window is closed immediately after the error message has been written. This happens because our mechanism for keeping a window alive was to wait for you to enter a character. However, in all three cases above, the program detected an error before it had read all of the characters, so that there was a character left on the input line. The program can’t tell such “leftover” characters from a character entered in response to the **Enter a character to close window** prompt. That “leftover” character then closed the window.

We could deal with that by modifying **main()** (see §5.6.3):

```
catch (runtime_error& e) {
    cerr << e.what() << '\n';
    // keep_window_open():
    cout << "Please enter the character ~ to close the window\n";
    for (char ch; cin >> ch; )      // keep reading until we find a ~
        if (ch=='~') return 1;
    return 1;
}
```

Basically, we replaced `keep_window_open()` with our own code. Note that we still have our problem if a `~` happens to be a character to be read after an error, but that's rather unlikely.

When we encountered this problem we wrote a version of `keep_window_open()` that takes a string as its argument and closes the window only when you enter that string after getting the prompt, so a simpler solution is

```
catch (runtime_error& e) {
    cerr << e.what() << '\n';
    keep_window_open("~~");
    return 1;
}
```

Now examples such as

```
+1
!1~~
0
```

will cause the calculator to give the proper error messages, then say

Please enter ~~ to exit

and not exit until you enter the string `~~`.

The calculator takes input from the keyboard. That makes testing tedious: each time we make an improvement, we have to type in a lot of test cases (yet again!) to make sure we haven't broken anything. It would be much better if we could store our test cases somewhere and run them with a single command. Some operating systems (notably Unix) make it trivial to get `cin` to read from a file without modifying the program, and similarly to divert the output from `cout` to a file. If that's not convenient, we must modify the program to use a file (see Chapter 10).

Now consider:

1+2; q

and

1+2 q

We would like both to print the result (3) and then exit the program. Curiously enough,

1+2 q

does that, but the apparently cleaner

```
1+2; q
```

elicits a **Primary expected** error. Where would we look for this error? In **main()** where ; and **q** are handled, of course. We added those “print” and “quit” commands rather quickly to get the calculator to work (§6.7). Now we are paying for that haste. Consider again:

```
double val = 0;
while (cin) {
    cout << "> ";
    Token t = ts.get();
    if (t.kind == 'q') break;
    if (t.kind == ';')
        cout << "=" << val << '\n';
    else
        ts.putback(t);
        val = expression();
}
```

If we find a semicolon, we straightaway proceed to call **expression()** without checking for **q**. The first thing that **expression()** does is to call **term()**, which first calls **primary()**, which finds **q**. The letter **q** isn’t a **Primary** so we get our error message. So, we should test for **q** after testing for a semicolon. While we were at it, we felt the need to simplify the logic a bit, so the complete **main()** reads

```
int main()
try
{
    while (cin) {
        cout << "> ";
        Token t = ts.get();
        while (t.kind == ';') t=ts.get(); // eat ;'
        if (t.kind == 'q') {
            keep_window_open();
            return 0;
        }
        ts.putback(t);
        cout << "=" << expression() << '\n';
    }
    keep_window_open();
    return 0;
}
```

```
catch (exception& e) {
    cerr << e.what() << '\n';
    keep_window_open("~~");
    return 1;
}
catch (...) {
    cerr << "exception \n";
    keep_window_open("~~");
    return 2;
}
```

This makes for reasonably robust error handling. So we can start considering what else we can do to improve the calculator.

7.4 Negative numbers

If you tested the calculator, you found that it couldn't handle negative numbers elegantly. For example, this is an error:

-1/2

We have to write

(0-1)/2

That's not acceptable.

Finding such problems during late debugging and testing is common. Only now do we have the opportunity to see what our design really does and get the feedback that allows us to refine our ideas. When planning a project, it is wise to try to preserve time and flexibility to benefit from the lessons we learn here. All too often, “release 1.0” is shipped without needed refinements because a tight schedule or a rigid project management strategy prevents “late” changes to the specification; “late” addition of “features” is especially dreaded. In reality, when a program is good enough for simple use by its designers but not yet ready to ship, it isn’t “late” in the development sequence; it’s the earliest time when we can benefit from solid experience with the program. A realistic schedule takes that into account.

In this case, we basically need to modify the grammar to allow unary minus. The simplest change seems to be in **Primary**. We have

Primary:
Number
"(Expression)"

and we need something like

```
Primary:
Number
"(" Expression ")"
"-" Primary
"+" Primary
```

We added unary plus because that's what C++ does. When we have unary minus, someone always tries unary plus and it's easier just to implement that than to explain why it is useless. The code that implements **Primary** becomes

```
double primary()
{
    Token t = ts.get();
    switch (t.kind) {
        case '(':      // handle '(' expression ')'
        {
            double d = expression();
            t = ts.get();
            if (t.kind != ')') error("')' expected");
            return d;
        }
        case '8':           // we use '8' to represent a number
            return t.value;      // return the number's value
        case '-':
            return - primary();
        case '+':
            return primary();
        default:
            error("primary expected");
    }
}
```

That's so simple that it actually worked the first time.

7.5 Remainder: %

When we first analyzed the ideals for a calculator, we wanted the remainder (modulo) operator: **%**. However, **%** is not defined for floating-point numbers, so we backed off. Now we can consider it again. It should be simple:

1. We add **%** as a **Token**.
2. We define a meaning for **%**.

We know the meaning of `%` for integer operands. For example:

```
> 2%3;
= 2
> 3%2;
= 1
> 5%3;
= 2
```

But how should we handle operands that are not integers? Consider:

```
> 6.7%3.3;
```

What should be the resulting value? There is no perfect technical answer. However, modulo is often defined for floating-point operands. In particular, `x%y` can be defined as `x-y=x-y*int(x/y)`, so that `6.7%3.3==6.7-3.3*int(6.7/3.3)`, that is, `0.1`. This is easily done using the standard library function `fmod()` (floating-point modulo) from `<cmath>` (§24.8). We modify `term()` to include

```
case '%':
{   double d = primary();
    if (d == 0) error("divide by zero");
    left = fmod(left,d);
    t = ts.get();
    break;
}
```

The `<cmath>` library is where we find all of the standard mathematical functions, such as `sqrt(x)` (square root of `x`), `abs(x)` (absolute value of `x`), `log(x)` (natural logarithm of `x`), and `pow(x,e)` (`x` to the power of `y`).

Alternatively, we can prohibit the use of `%` on a floating-point argument. We check if the floating-point operands have fractional parts and give an error message if they do. The problem of ensuring `int` operands for `%` is a variant of the narrowing problem (§3.9.2 and §5.6.4), so we could solve it using `narrow_cast`:

```
case '%':
{   int i1 = narrow_cast<int>(left);
    int i2 = narrow_cast<int>(primary());
    if (i2 == 0) error("%: divide by zero");
    left = i1%i2;
    t = ts.get();
    break;
}
```

For a simple calculator, either solution will do.

7.6 Cleaning up the code



We have made several changes to the code. They are, we think, all improvements, but the code is beginning to look a bit messy. Now is a good time to review the code to see if we can make it clearer and shorter, add and improve comments, etc. In other words, we are not finished with the program until we have it in a state suitable for someone else to take over maintenance. Except for the almost total absence of comments, the calculator code really isn't that bad, but let's do a bit of cleanup.

7.6.1 Symbolic constants

Looking back, we find the use of '**8**' to indicate a **Token** containing a numeric value odd. It doesn't really matter what value is used to indicate a number **Token** as long as the value is distinct from all other values indicating different kinds of **Tokens**. However, the code looks a bit odd and we had to keep reminding ourselves in comments:

```
case '8':           // we use '8' to represent a number
    return t.value;   // return the number's value
case '-':
    return - primary();
```



To be honest, we also made a few mistakes, typing '**0**' rather than '**8**', because we forgot which value we had chosen to use. In other words, using '**8**' directly in the code manipulating **Tokens** was sloppy, hard to remember, and error-prone; '**8**' is one of those “magic constants” we warned against in §4.3.1. What we should have done was to introduce a symbolic name for the constant we used to represent a number:

```
const char number = '8'; // t.kind==number means that t is a number Token
```

The **const** modifier simply tells the compiler that we are defining an object that is not supposed to change: for example, an assignment **number='0'** would cause the compiler to give an error message. Given that definition of **number**, we don't have to use '**8**' explicitly anymore. The code fragment from **primary** above now becomes

```
case number:
    return t.value;           // return the number's value
case '-':
    return - primary();
```



This requires no comment. We should not say in comments what can be clearly and directly said in code. Repeated comments explaining something are often an indication that the code should be improved.

Similarly, the code in `Token_stream::get()` that recognizes numbers becomes

```
case '!':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
{   cin.putback(ch);    // put digit back into the input stream
    double val;
    cin >> val;        // read a floating-point number
    return Token(number,val);
}
```

We could consider symbolic names for all tokens, but that seems overkill. After all, '`(`' and '`+`' are about as obvious a notation for `(` and `+` as anyone could come up with. Looking through the tokens, only '`:`' for "print" (or "terminate expression") and '`q`' for "quit" seem arbitrary. Why not '`p`' and '`e`'? In a larger program, it is only a matter of time before such obscure and arbitrary notation becomes a cause of a problem, so we introduce

```
const char quit = 'q';           // t.kind==quit means that t is a quit Token
const char print = ':';          // t.kind==print means that t is a print Token
```

Now we can write `main()`'s loop like this:

```
while (cin) {
    cout << "> ";
    Token t = ts.get();
    while (t.kind == print) t=ts.get();
    if (t.kind == quit) {
        keep_window_open();
        return 0;
    }
    ts.putback(t);
    cout << "=" << expression() << '\n';
}
```

Introducing symbolic names for "print" and "quit" makes the code easier to read. In addition, it doesn't encourage someone reading `main()` to make assumptions about how "print" and "quit" are represented on input. For example, it should come as no surprise if we decide to change the representation of "quit" to '`e`' (for "exit"). That would now require no change in `main()`.

Now the strings "`>`" and "`=`" stand out. Why do we have these "magical" literals in the code? How would a new programmer reading `main()` guess their

purpose? Maybe we should add a comment? Adding a comment might be a good idea, but introducing a symbolic name is more effective:

```
const string prompt = "> ";
const string result = "=";      // used to indicate that what follows is a result
```

Should we want to change the prompt or the result indicator, we can just modify those **consts**. The loop now reads

```
while (cin {
    cout << prompt;
    Token t = ts.get();
    while (t.kind == print) t=ts.get();
    if (t.kind == quit) {
        keep_window_open();
        return 0;
    }
    ts.putback(t);
    cout << result << expression() << '\n';
}
}
```

7.6.2 Use of functions

The functions we use should reflect the structure of our program, and the names of the functions should identify the logically separate parts of our code. Basically, our program so far is rather good in this respect: **expression()**, **term()**, and **primary()** directly reflect our understanding of the expression grammar, and **get()** handles the input and token recognition. Looking at **main()**, though, we notice that it does two logically separate things:

1. **main()** provides general “scaffolding”: start the program, end the program, and handle “fatal” errors.
2. **main()** handles the calculation loop.



Ideally, a function performs a single logical action (§4.5.1). Having **main()** perform both of these actions obscures the structure of the program. The obvious solution is to make the calculation loop into a separate function **calculate()**:

```
void calculate()      // expression evaluation loop
{
    while (cin {
        cout << prompt;
        Token t = ts.get();
        while (t.kind == print) t=ts.get();      // first discard all "prints"
```

```

        if (t.kind == quit) return;
        ts.putback(t);
        cout << result << expression() << '\n';
    }
}

int main()
try {
    calculate();
    keep_window_open();      // cope with Windows console mode
    return 0;
}
catch (runtime_error& e) {
    cerr << e.what() << '\n';
    keep_window_open("~~");
    return 1;
}
catch (...) {
    cerr << "exception \n";
    keep_window_open("~~");
    return 2;
}

```

This reflects the structure much more directly and is therefore easier to understand.

7.6.3 Code layout

Looking through the code for ugly code, we find

```

switch (ch) {
case 'q': case ';': case '%': case '(': case ')': case '+': case '-': case '*': case '/':
    return Token{ch};           // let each character represent itself
}

```

This wasn't too bad before we added '**'q'**', '**';'**', and '**'%'**', but now it's beginning to become obscure. Code that is hard to read is where bugs can more easily hide. And yes, a potential bug lurks here! Using one line per case and adding a couple of comments help. So, **Token_stream**'s **get()** becomes

```

Token Token_stream::get()
    // read characters from cin and compose a Token
{
    if (full) {    // check if we already have a Token ready
        full = false;
        return buffer;
    }
}

```

```
char ch;
cin >> ch;      // note that >> skips whitespace (space, newline, tab, etc.)

switch (ch) {
    case quit:
    case print:
    case '(':
    case ')':
    case '+':
    case '-':
    case '*':
    case '/':
    case '%':
        return Token{ch};      // let each character represent itself
    case '.':           // a floating-point-literal can start with a dot
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':   // numeric literal
    {   cin.putback(ch);          // put digit back into the input stream
        double val;
        cin >> val;            // read a floating-point number
        return Token{number,val};
    }
    default:
        error("Bad token");
}
}
```

We could of course have put each digit case on a separate line also, but that didn't seem to buy us any clarity. Also, doing so would prevent `get()` from being viewed in its entirety on a screen at once. Our ideal is for each function to fit on the screen; one obvious place for a bug to hide is in the code that we can't see because it's off the screen horizontally or vertically. Code layout matters.

Note also that we changed the plain '`q`' to the symbolic name `quit`. This improves readability and also guarantees a compile-time error if we should make the mistake of choosing a value for `quit` that clashes with another token name.

When we clean up code, we might accidentally introduce errors. Always re-test the program after cleanup. Better still, do a bit of testing after each set of minor improvements so that if something went wrong you can still remember exactly what you did. Remember: Test early and often.



7.6.4 Commenting

We added a few comments as we went along. Good comments are an important part of writing code. We tend to forget about comments in the heat of programming. When you go back to the code to clean it up is an excellent time to look at each part of the program to see if the comments you originally wrote are

1. Still valid (you might have changed the code since you wrote the comment)
2. Adequate for a reader (they usually are not)
3. Not so verbose that they distract from the code

To emphasize that last concern: what is best said in code should be said in code. Avoid comments that explain something that's perfectly clear to someone who knows the programming language. For example:

x = b+c; // add b and c and assign the result to x

You'll find such comments in this book, but only when we are trying to explain the use of a language feature that might not yet be familiar to you.

Comments are for things that code expresses poorly. An example is intent: code says what it does, not what it was intended to do (§5.9.1). Look at the calculator code. There is something missing: the functions show how we process expressions and tokens, but there is no indication (except the code) of what we meant expressions and tokens to be. The grammar is a good candidate for something to put in comments or into some documentation of the calculator.

```
/*
Simple calculator
```

Revision history:

Revised by Bjarne Stroustrup November 2013

Revised by Bjarne Stroustrup May 2007

Revised by Bjarne Stroustrup August 2006

Revised by Bjarne Stroustrup August 2004

*Originally written by Bjarne Stroustrup
(bs@cs.tamu.edu) Spring 2004.*

This program implements a basic expression calculator.

Input from cin; output to cout.

The grammar for input is:

Statement:

Expression
Print
Quit

Print:

;

Quit:

q

Expression:

Term
Expression + Term
Expression – Term

Term:

Primary
*Term * Primary*
Term / Primary
Term % Primary

Primary:

Number
(Expression)
- Primary
+ Primary

Number:

floating-point-literal

Input comes from cin through the Token_stream called ts.

**/*

Here we used the block comment, which starts with a `/*` and continues until a `*/`. In a real program, the revision history would contain indications of what corrections and improvements were made.

Note that the comments are not the code. In fact, this grammar simplifies a bit: compare the rule for **Statement** with what really happens (e.g., have a peek at the code in the following section). The comment fails to explain the loop in **calculate()** that allows us to do several calculations in a single run of the program. We'll return to that problem in §7.8.1.

7.7 Recovering from errors

Why do we exit when we find an error? That seemed simple and obvious at the time, but why? Couldn't we just write an error message and carry on? After all, we often make little typing errors and such an error doesn't mean that we have decided not to do a calculation. So let's try to recover from an error. That basically means that we have to catch exceptions and continue after we have cleaned up any messes that were left behind.

Until now, all errors have been represented as exceptions and handled by `main()`. If we want to recover from errors, `calculate()` must catch exceptions and try to clean up the mess before trying to evaluate the next expression:

```
void calculate()
{
    while (cin)
        try {
            cout << prompt;
            Token t = ts.get();
            while (t.kind == print) t=ts.get();      // first discard all "prints"
            if (t.kind == quit) return;
            ts.putback(t);
            cout << result << expression() << '\n';
        }
        catch (exception& e) {
            cerr << e.what() << '\n';                // write error message
            clean_up_mess();
        }
}
```

We simply made the `while`-loop's block into a `try`-block that writes an error message and cleans up the mess. Once that's done, we carry on as always.

What would “clean up the mess” entail? Basically, getting ready to compute again after an error has been handled means making sure that all our data is in a good and predictable state. In the calculator, the only data we keep outside an individual function is the `Token_stream`. So what we need to do is to ensure that we don't have tokens related to the aborted calculation sitting around to confuse the next calculation. For example,

```
1++2*3; 4+5;
```

will cause an error, and **2*3; 4+5** will be left in the **Token_stream**'s and **cin**'s buffers after the second **+** has triggered an exception. We have two choices:

1. Purge all tokens from the **Token_stream**.
2. Purge all tokens from the current calculation from the **Token_stream**.

The first choice discards all (including **4+5;**), whereas the second choice just discards **2*3;**, leaving **4+5** to be evaluated. Either could be a reasonable choice and either could surprise a user. As it happens, both are about equally simple to implement. We chose the second alternative because it simplifies testing.

So we need to read input until we find a semicolon. This seems simple. We have **get()** to do our reading for us so we can write a **clean_up_mess()** like this:

```
void clean_up_mess()           // naive
{
    while (true) {             // skip until we find a print
        Token t = ts.get();
        if (t.kind == print) return;
    }
}
```

Unfortunately, that doesn't work all that well. Why not? Consider this input:

1@z; 1+3;

The **@** gets us into the **catch**-clause for the **while**-loop. Then, we call **clean_up_mess()** to find the next semicolon. Then, **clean_up_mess()** calls **get()** and reads the **z**. That gives another error (because **z** is not a token) and we find ourselves in **main()**'s **catch(...)** handler, and the program exits. Oops! We don't get a chance to evaluate **1+3**. Back to the drawing board!

We could try more elaborate **trys** and **catches**, but basically we are heading into an even bigger mess. Errors are hard to handle, and errors during error handling are even worse than other errors. So, let's try to devise some way to flush characters out of a **Token_stream** that couldn't possibly throw an exception. The only way of getting input into our calculator is **get()**, and that can – as we just discovered the hard way – throw an exception. So we need a new operation. The obvious place to put that is in **Token_stream**:

```
class Token_stream {
public:
    Token get();           // get a Token
    void putback(Token t); // put a Token back
    void ignore(char c);   // discard characters up to and including a c
```

```
private:
    bool full {false};      // is there a Token in the buffer?
    Token buffer;          // here is where we keep a Token put back using
                           // putback()
};
```

This **ignore()** function needs to be a member of **Token_stream** because it needs to look at **Token_stream**'s buffer. We chose to make “the thing to look for” an argument to **ignore()** – after all, the **Token_stream** doesn't have to know what the calculator considers a good character to use for error recovery. We decided that argument should be a character because we don't want to risk composing **Tokens** – we saw what happened when we tried that. So we get

```
void Token_stream::ignore(char c)
    // c represents the kind of Token
{
    // first look in buffer:
    if (full && c==buffer.kind) {
        full = false;
        return;
    }
    full = false;

    // now search input:
    char ch = 0;
    while (cin>>ch)
        if (ch==c) return;
}
```

This code first looks at the buffer. If there is a **c** there, we are finished after discarding that **c**; otherwise, we need to read characters from **cin** until we find a **c**.

We can now write **clean_up_mess()** rather simply:

```
void clean_up_mess()
{
    ts.ignore(print);
}
```

Dealing with errors is always tricky. It requires much experimentation and testing because it is extremely hard to imagine what errors can occur. Trying to make a program foolproof is always a very technical activity; amateurs typically don't care. Quality error handling is one mark of a professional.

7.8 Variables

Having worked on style and error handling, we can return to looking for improvements in the calculator functionality. We now have a program that works quite well; how can we improve it? The first wish list for the calculator included variables. Having variables gives us better ways of expressing longer calculations. Similarly, for scientific calculations, we'd like built-in named values, such as **pi** and **e**, just as we have on scientific calculators.

Adding variables and constants is a major extension to the calculator. It will touch most parts of the code. This is the kind of extension that we should not embark on without good reason and sufficient time. Here, we add variables and constants because it gives us a chance to look over the code again and try out some more programming techniques.

7.8.1 Variables and definitions

Obviously, the key to both variables and built-in constants is for the calculator program to keep *(name,value)* pairs so that we can access the value given the name. We can define a **Variable** like this:

```
class Variable {  
public:  
    string name;  
    double value;  
};
```

We will use the **name** member to identify a **Variable** and the **value** member to store the value corresponding to that **name**.

How can we store **Variables** so that we can search for a **Variable** with a given **name** string to find its value or to give it a new value? Looking back over the programming tools we have encountered so far, we find only one good answer: a **vector** of **Variables**:

```
vector<Variable> var_table;
```

We can put as many **Variables** as we like into the vector **var_table** and search for a given name by looking at the vector elements one after another. We can write a **get_value()** function that looks for a given **name** string and returns its corresponding **value**:

```
double get_value(string s)  
    // return the value of the Variable named s  
{
```

```

for (const Variable& v : var_table)
    if (v.name == s) return v.value;
    error("get: undefined variable ", s);
}

```

The code really is quite simple: go through every **Variable** in **var_table** (starting with the first element and continuing until the last) and see if its **name** matches the argument string **s**. If that is the case, return its **value**.

Similarly, we can define a **set_value()** function to give a **Variable** a new **value**:

```

void set_value(string s, double d)
    // set the Variable named s to d
{
    for (Variable& v : var_table)
        if (v.name == s) {
            v.value = d;
            return;
        }
    error("set: undefined variable ", s);
}

```

We can now read and write “variables” represented as **Variables** in **var_table**. How do we get a new **Variable** into **var_table**? What does a user of our calculator have to write to define a new variable and later to get its value? We could consider C++’s notation

```
double var = 7.2;
```

That would work, but all variables in this calculator hold **double** values, so saying “double” would be redundant. Could we make do with

```
var = 7.2;
```

Possibly, but then we would be unable to tell the difference between the declaration of a new variable and a spelling mistake:

```

var1 = 7.2;    // define a new variable called var1
var1 = 3.2;    // define a new variable called var2

```

Oops! Clearly, we meant **var2 = 3.2**; but we didn’t say so (except in the comment). We could live with this, but we’ll follow the tradition in languages, such as C++, that distinguish declarations (with initializations) from assignments. We could use **double**, but for a calculator we’d like something short, so – drawing on another old tradition – we choose the keyword **let**:

```
let var = 7.2;
```

The grammar would be

```
Calculation:
  Statement
    Print
    Quit
  Calculation Statement

Statement:
  Declaration
  Expression

Declaration:
  "let" Name "=" Expression
```

Calculation is the new top production (rule) of the grammar. It expresses the loop (in **calculate()**) that allows us to do several calculations in a run of the calculator program. It relies on the **Statement** production to handle expressions and declarations. We can handle a statement like this:

```
double statement()
{
    Token t = ts.get();
    switch (t.kind) {
        case let:
            return declaration();
        default:
            ts.putback(t);
            return expression();
    }
}
```

We can now use **statement()** instead of **expression()** in **calculate()**:

```
void calculate()
{
    while (cin)
        try {
            cout << prompt;
            Token t = ts.get();
            while (t.kind == print) t=ts.get();    // first discard all "prints"
            if (t.kind == quit) return;           // quit
        }
}
```

```

        ts.putback(t);
        cout << result << statement() << '\n';
    }
    catch (exception& e) {
        cerr << e.what() << '\n';           // write error message
        clean_up_mess();
    }
}

```

We now have to write `declaration()`. What should it do? It should make sure that what comes after a `let` is a **Name** followed by a `=` followed by an **Expression**. That's what our grammar says. What should it do with the **name**? We should add a **Variable** with that **name** string and the value of the expression to our `vector<Variable>` called `var_table`. Once that's done we can retrieve the value using `get_value()` and change it using `set_value()`. However, before writing this, we have to decide what should happen if we define a variable twice. For example:

```

let v1 = 7;
let v1 = 8;

```

We chose to consider such a redefinition an error. Typically, it is simply a spelling mistake. Instead of what we wrote, we probably meant

```

let v1 = 7;
let v2 = 8;

```

There are logically two parts to defining a **Variable** with the name **var** with the value **val**:

1. Check whether there already is a **Variable** called **var** in `var_table`.
2. Add `(var, val)` to `var_table`.

We have no use for uninitialized variables. We defined the functions `is_declared()` and `define_name()` to represent those two logically separate operations:

```

bool is_declared(string var)
    // is var already in var_table?
{
    for (const Variable& v : var_table)
        if (v.name == var) return true;
    return false;
}

```

```
double define_name(string var, double val)
    // add (var,val) to var_table
{
    if (is_declared(var)) error(var, " declared twice");
    var_table.push_back(Variable(var, val));
    return val;
}
```

Adding a new **Variable** to a **vector<Variable>** is easy; that's what **vector**'s **push_back()** member function does:

```
var_table.push_back(Variable(var, val));
```

The **Variable(var, val)** makes the appropriate **Variable** and **push_back()**, then adds that **Variable** to the end of **var_table**. Given that, and assuming that we can handle **let** and **name** tokens, **declaration()** is straightforward to write:

```
double declaration()
    // assume we have seen "let"
    // handle: name = expression
    // declare a variable called "name" with the initial value "expression"
{
    Token t = ts.get();
    if (t.kind != name) error ("name expected in declaration");
    string var_name = t.name;

    Token t2 = ts.get();
    if (t2.kind != '=') error("= missing in declaration of ", var_name);

    double d = expression();
    define_name(var_name, d);
    return d;
}
```

Note that we returned the value stored in the new variable. That's useful when the initializing expression is nontrivial. For example:

```
let v = d/(t2-t1);
```

This declaration will define **v** and also print its value. Additionally, printing the value of a declared variable simplifies the code in **calculate()** because every **statement()** returns a value. General rules tend to keep code simple, whereas special cases tend to lead to complications.

This mechanism for keeping track of **Variables** is what is often called a *symbol table* and could be radically simplified by the use of a standard library **map**; see §21.6.1.

7.8.2 Introducing names

This is all very good, but unfortunately, it doesn't quite work. By now, that shouldn't come as a surprise. Our first cut never – well, hardly ever – works. Here, we haven't even finished the program – it doesn't yet compile. We have no '`=`' token, but that's easily handled by adding a case to **Token_stream::get()** (§7.6.3). But how do we represent **let** and **name** as tokens? Obviously, we need to modify **get()** to recognize these tokens. How? Here is one way:

```
const char name = 'a';           // name token
const char let = 'L';            // declaration token
const string declkey = "let";    // declaration keyword

Token Token_stream::get()
{
    if (full) {
        full = false;
        return buffer;
    }
    char ch;
    cin >> ch;
    switch (ch) {
        // as before
    default:
        if (isalpha(ch)) {
            cin.putback(ch);
            string s;
            cin>>s;
            if (s == declkey) return Token(let); // declaration keyword
            return Token{name,s};
        }
        error("Bad token");
    }
}
```

Note first of all the call **isalpha(ch)**. This call answers the question “Is **ch** a letter?”, **isalpha()** is part of the standard library that we get from **std_lib_facilities.h**. For more character classification functions, see §11.6. The logic for recognizing names is the same as that for recognizing numbers: find a first character of the

right kind (here, a letter), then put it back using `putback()` and read in the whole name using `>>`.

Unfortunately, this doesn't compile; we have no `Token` that can hold a `string`, so the compiler rejects `Token{name,s}`. To handle that, we must modify the definition of `Token` to hold either a `string` or a `double`, and handle three forms of initializers, such as

- Just a `kind`; for example, `Token{!*}`
- A `kind` and a number; for example, `Token{number,4.321}`
- A `kind` and a `name`; for example, `Token{name,"pi"}`

We handle that by introducing three initialization functions, known as constructors because they construct objects:

```
class Token {
public:
    char kind;
    double value;
    string name;
    Token(char ch) :kind{ch} {}           // initialize kind with ch
    Token(char ch, double val) :kind{ch}, value{val} {} // initialize kind
                                            // and value
    Token(char ch, string n) :kind{ch}, name{n} {}      // initialize kind
                                            // and name
};
```

Constructors add an important degree of control and flexibility to initialization. We will examine constructors in detail in Chapter 9 (§9.4.2, §9.7).

We chose '`L`' as the representation of the `let` token and the string `let` as our keyword. Obviously, it would be trivial to change that keyword to `double`, `var`, `#`, or whatever by changing the string `deckey` that we compare `s` to.

Now we try the program again. If you type this, you'll see that it all works:

```
let x = 3.4;
let y = 2;
x + y * 2;
```

However, this doesn't work:

```
let x = 3.4;
let y = 2;
x+y*2;
```

What's the difference between those two examples? Have a look to see what happens.

The problem is that we were sloppy with our definition of **Name**. We even “forgot” to define our **Name** production in the grammar (§7.8.1). What characters can be part of a name? Letters? Certainly. Digits? Certainly, as long as they are not the starting character. Underscores? Eh? The `+` character? Well? Eh? Look at the code again. After the initial letter we read into a **string** using `>>`. That accepts every character until it sees whitespace. So, for example, `x+y*2;` is a single name – even the trailing semicolon is read as part of the name. That's unintended and unacceptable.

What must we do instead? First we must specify precisely what we want a name to be, and then we must modify **get()** to do that. Here is a workable specification of a name: a sequence of letters and digits starting with a letter. Given this definition,

```
a
ab
a1
Z12
asdsddsfdfdasfdasf434RTHTD12345dfdsa8fsd888fadsf
```

are names and

```
1a
as_s
#
as*
a car
```

are not. Except for leaving out the underscore, this is C++'s rule. We can implement that in the default case of **get()**:

```
default:
if (isalpha(ch)) {
    string s;
    s += ch;
    while (cin.get(ch) && (isalpha(ch) || isdigit(ch))) s+=ch;
    cin.putback(ch);
    if (s == declkey) return Token{let}; // declaration keyword
    return Token{name,s};
}
error("Bad token");
```

Instead of reading directly into the **string s**, we read characters and put those into **s** as long as they are letters or digits. The **s+=ch** statement adds (appends) the character **ch** to the end of the string **s**. The curious statement

```
while (cin.get(ch) && (isalpha(ch) || isdigit(ch))) s+=ch;
```

reads a character into **ch** (using **cin**'s member function **get()**) and checks if it is a letter or a digit. If so, it adds **ch** to **s** and reads again. The **get()** member function works just like **>>** except that it doesn't by default skip whitespace.

7.8.3 Predefined names

Now that we have names, we can easily predefine a few common ones. For example, if we imagine that our calculator will be used for scientific calculations, we'd want **pi** and **e**. Where in the code would we define those? In **main()** before the call of **calculate()** or in **calculate()** before the loop. We'll put them in **main()** because those definitions really aren't part of any calculation:

```
int main()
try {
    // predefined names:
    define_name("pi",3.1415926535);
    define_name("e",2.7182818284);

    calculate();

    keep_window_open();      // cope with Windows console mode
    return 0;
}
catch (exception& e) {
    cerr << e.what() << '\n';
    keep_window_open("~/");
    return 1;
}
catch (...) {
    cerr << "exception \n";
    keep_window_open("~/");
    return 2;
}
```

7.8.4 Are we there yet?

Not really. We have made so many changes that we need to test everything again, clean up the code, and review the comments. Also, we could do more definitions. For example, we "forgot" to provide an assignment operator (see exercise 2), and

if we have an assignment we might want to distinguish between variables and constants (exercise 3).

Initially, we backed off from having named variables in our calculator. Looking back over the code that implements them, we may have two possible reactions:

1. Implementing variables wasn't all that bad; it took only about three dozen lines of code.
2. Implementing variables was a major extension. It touched just about every function and added a completely new concept to the calculator. It increased the size of the calculator by 45% and we haven't even implemented assignment!

In the context of a first program of significant complexity, the second reaction is the correct one. More generally, it's the right reaction to any suggestion that adds something like 50% to a program in terms of both size and complexity. When that has to be done, it is more like writing a new program based on a previous one than anything else, and it should be treated that way. In particular, if you can build a program in stages as we did with the calculator, and test it at each stage, you are far better off doing so than trying to do the whole program all at once.



Drill

1. Starting from the file `calculator08buggy.cpp`, get the calculator to compile.
2. Go through the entire program and add appropriate comments.
3. As you commented, you found errors (deviously inserted especially for you to find). Fix them; they are not in the text of the book.
4. Testing: prepare a set of inputs and use them to test the calculator. Is your list pretty complete? What should you look for? Include negative values, 0, very small, very large, and “silly” inputs.
5. Do the testing and fix any bugs that you missed when you commented.
6. Add a predefined name `k` meaning `1000`.
7. Give the user a square root function `sqrt()`, for example, `sqrt(2+6.7)`. Naturally, the value of `sqrt(x)` is the square root of `x`; for example, `sqrt(9)` is `3`. Use the standard library `sqrt()` function that is available through the header `std_lib_facilities.h`. Remember to update the comments, including the grammar.
8. Catch attempts to take the square root of a negative number and print an appropriate error message.
9. Allow the user to use `pow(x,i)` to mean “Multiply `x` with itself `i` times”; for example, `pow(2.5,3)` is `2.5*2.5*2.5`. Require `i` to be an integer using the technique we used for `%`.

10. Change the “declaration keyword” from `let` to `#`.
11. Change the “quit keyword” from `quit` to `exit`. That will involve defining a string for `quit` just as we did for `let` in §7.8.2.

Review

1. What is the purpose of working on the program after the first version works? Give a list of reasons.
2. Why does `1+2; q` typed into the calculator not quit after it receives an error?
3. Why did we choose to make a constant character called `number`?
4. We split `main()` into two separate functions. What does the new function do and why did we split `main()`?
5. Why do we split code into multiple functions? State principles.
6. What is the purpose of commenting and how should it be done?
7. What does `narrow_cast` do?
8. What is the use of symbolic constants?
9. Why do we care about code layout?
10. How do we handle `%` (remainder) of floating-point numbers?
11. What does `is_declared()` do and how does it work?
12. The input representation for `let` is more than one character. How is it accepted as a single token in the modified code?
13. What are the rules for what names can and cannot be in the calculator program?
14. Why is it a good idea to build a program incrementally?
15. When do you start to test?
16. When do you retest?
17. How do you decide what should be a separate function?
18. How do you choose names for variables and functions? List possible reasons.
19. Why do you add comments?
20. What should be in comments and what should not?
21. When do we consider a program finished?

Terms

code layout	maintenance	scaffolding
commenting	recovery	symbolic constant
error handling	revision history	testing
feature creep		

Exercises

1. Allow underscores in the calculator's variable names.
2. Provide an assignment operator, `=`, so that you can change the value of a variable after you introduce it using `let`. Discuss why that can be useful and how it can be a source of problems.
3. Provide named constants that you really can't change the value of. Hint: You have to add a member to `Variable` that distinguishes between constants and variables and check for it in `set_value()`. If you want to let the user define constants (rather than just having `pi` and `e` defined as constants), you'll have to add a notation to let the user express that, for example, `const pi = 3.14;`.
4. The `get_value()`, `set_value()`, `is_declared()`, and `define_name()` functions all operate on the variable `var_table`. Define a class called `Symbol_table` with a member `var_table` of type `vector<Variable>` and member functions `get()`, `set()`, `is_declared()`, and `declare()`. Rewrite the calculator to use a variable of type `Symbol_table`.
5. Modify `Token_stream::get()` to return `Token(print)` when it sees a new-line. This implies looking for whitespace characters and treating newline (`\n!`) specially. You might find the standard library function `isspace(ch)`, which returns `true` if `ch` is a whitespace character, useful.
6. Part of what every program should do is to provide some way of helping its user. Have the calculator print out some instructions for how to use the calculator if the user presses the H key (both upper- and lowercase).
7. Change the `q` and `h` commands to be `quit` and `help`, respectively.
8. The grammar in §7.6.4 is incomplete (we did warn you against overreliance on comments); it does not define sequences of statements, such as `4+4; 5-6;;`, and it does not incorporate the grammar changes outlined in §7.8. Fix that grammar. Also add whatever you feel is needed to that comment as the first comment of the calculator program and its overall comment.
9. Suggest three improvements (not mentioned in this chapter) to the calculator. Implement one of them.
10. Modify the calculator to operate on `ints` (only); give errors for overflow and underflow. Hint: Use `narrow_cast` (§7.5).
11. Revisit two programs you wrote for the exercises in Chapter 4 or 5. Clean up that code according to the rules outlined in this chapter. See if you find any bugs in the process.

Postscript

As it happens, we have now seen a simple example of how a compiler works. The calculator analyzes input broken down into tokens and understood according to a grammar. That's exactly what a compiler does. After analyzing its input, a compiler then produces a representation (object code) that we can later execute. The calculator immediately executes the expressions it has analyzed; programs that do this are called interpreters rather than compilers.



Technicalities: Functions, etc.

“No amount of genius can overcome obsession with detail.”

—Traditional

In this chapter and the next, we change our focus from programming to our main tool for programming: the C++ programming language. We present language-technical details to give a slightly broader view of C++’s basic facilities and to provide a more systematic view of those facilities. These chapters also act as a review of many of the programming notions presented so far and provide an opportunity to explore our tool without adding new programming techniques or concepts.

8.1 Technicalities	8.5.4 Pass-by-<code>const</code>-reference
8.2 Declarations and definitions	8.5.5 Pass-by-reference
8.2.1 Kinds of declarations	8.5.6 Pass-by-value vs. pass-by-reference
8.2.2 Variable and constant declarations	8.5.7 Argument checking and conversion
8.2.3 Default initialization	8.5.8 Function call implementation
8.3 Header files	8.5.9 <code>constexpr</code> functions
8.4 Scope	
8.5 Function call and return	8.6 Order of evaluation
8.5.1 Declaring arguments and return type	8.6.1 Expression evaluation
8.5.2 Returning a value	8.6.2 Global initialization
8.5.3 Pass-by-value	
	8.7 Namespaces
	8.7.1 <code>using</code> declarations and <code>using</code> directives

8.1 Technicalities

Given a choice, we'd much rather talk about programming than about programming language features; that is, we consider how to express ideas as code far more interesting than the technical details of the programming language that we use to express those ideas. To pick an analogy from natural languages: we'd much rather discuss the ideas in a good novel and the way those ideas are expressed than study the grammar and vocabulary of English. What matters are ideas and how those ideas can be expressed in code, not the individual language features.

However, we don't always have a choice. When you start programming, your programming language is a foreign language for which you need to look at "grammar and vocabulary." This is what we will do in this chapter and the next, but please don't forget:

- Our primary study is programming.
- Our output is programs/systems.
- A programming language is (only) a tool.

Keeping this in mind appears to be amazingly difficult. Many programmers come to care passionately about apparently minor details of language syntax and semantics. In particular, too many get the mistaken belief that the way things are done in their first programming language is "the one true way." Please don't fall into that trap. C++ is in many ways a very nice language, but it is not perfect; neither is any other programming language.

Most design and programming concepts are universal, and many such concepts are widely supported by popular programming languages. That means that the fundamental ideas and techniques we learn in a good programming course carry over from language to language. They can be applied – with varying degrees of ease – in all languages. The language technicalities, however, are specific

to a given language. Fortunately, programming languages do not develop in a vacuum, so much of what you learn here will have reasonably obvious counterparts in other languages. In particular, C++ belongs to a group of languages that also includes C (Chapter 27), Java, and C#, so quite a few technicalities are shared with those languages.

Note that when we are discussing language-technical issues, we deliberately use nondescriptive names, such as **f**, **g**, **X**, and **y**. We do that to emphasize the technical nature of such examples, to keep those examples very short, and to try to avoid confusing you by mixing language technicalities and genuine program logic. When you see nondescriptive names (such as `should never be used in real code`), please focus on the language-technical aspects of the code. Technical examples typically contain code that simply illustrates language rules. If you compiled and ran them, you'd get many “variable not used” warnings, and few such technical program fragments would do anything sensible.

Please note that what we write here is not a complete description of C++'s syntax and semantics – not even for the facilities we describe. The ISO C++ standard is 1300+ pages of dense technical language and *The C++ Programming Language* by Stroustrup is 1300+ pages of text aimed at experienced programmers (both covering both the C++ language and its standard library). We do not try to compete with those in completeness and comprehensiveness; we compete with them in comprehensibility and value for time spent reading.

8.2 Declarations and definitions

A *declaration* is a statement that introduces a name into a scope (§8.4)

- Specifying a type for what is named (e.g., a variable or a function)
- Optionally, specifying an initializer (e.g., an initializer value or a function body)

For example:

```
int a = 7;           // an int variable
const double cd = 8.7; // a double-precision floating-point constant
double sqrt(double); // a function taking a double argument
                      // and returning a double result
vector<Token> v;   // a vector-of-Tokens variable
```

Before a name can be used in a C++ program, it must be declared. Consider:

```
int main()
{
    cout << f(i) << '\n';
}
```

The compiler will give at least three “undeclared identifier” errors for this: `cout`, `f`, and `i` are not declared anywhere in this program fragment. We can get `cout` declared by including the header `std_lib_facilities.h`, which contains its declaration:

```
#include "std_lib_facilities.h"      // we find the declaration of cout in here

int main()
{
    cout << f(i) << '\n';
}
```

Now, we get only two “undefined” errors. As you write real-word programs, you’ll find that most declarations are found in headers. That’s where we define interfaces to useful facilities defined “elsewhere.” Basically, a declaration defines how something can be used; it defines the interface of a function, variable, or class. Please note one obvious but invisible advantage of this use of declarations: we didn’t have to look at the details of how `cout` and its `<<` operators were defined; we just `#included` their declarations. We didn’t even have to look at their declarations; from textbooks, manuals, code examples, or other sources, we just know how `cout` is supposed to be used. The compiler reads the declarations in the header that it needs to “understand” our code.

However, we still have to declare `f` and `i`. We could do that like this:

```
#include "std_lib_facilities.h"      // we find the declaration of cout in here

int f(int);                      // declaration of f

int main()
{
    int i = 7;                    // declaration of i
    cout << f(i) << '\n';
}
```

This will compile because every name has been declared, but it will not link (§2.4) because we have not defined `f()`; that is, nowhere have we specified what `f()` actually does.

A declaration that (also) fully specifies the entity declared is called a *definition*. For example:

```
int a = 7;
vector<double> v;
double sqrt(double d) /* . . . */
```

Every definition is (by definition ☺) also a declaration, but only some declarations are also definitions. Here are some examples of declarations that are not definitions; if the entity it refers to is used, each must be matched by a definition elsewhere in the code:

```
double sqrt(double);    // no function body here  
extern int a;           // "extern plus no initializer" means "not definition"
```

When we contrast definitions and declarations, we follow convention and use *declarations* to mean “declarations that are not definitions” even though that’s slightly sloppy terminology.

A definition specifies exactly what a name refers to. In particular, a definition of a variable sets aside memory for that variable. Consequently, you can’t define something twice. For example:

```
double sqrt(double d) /* ... */ // definition  
double sqrt(double d) /* ... */ // error: double definition  
  
int a;                  // definition  
int a;                  // error: double definition
```

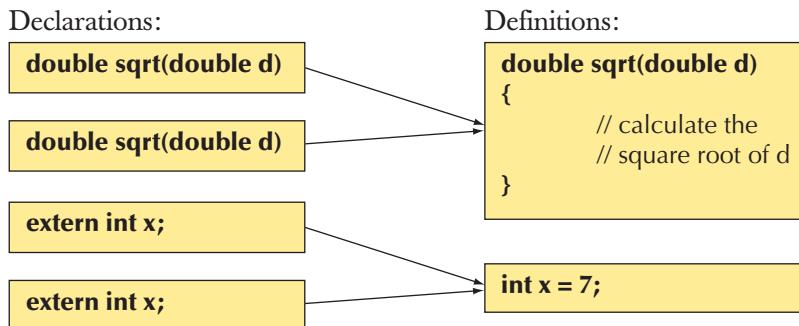
In contrast, a declaration that isn’t also a definition simply tells how you can use a name; it is just an interface and doesn’t allocate memory or specify a function body. Consequently, you can declare something as often as you like as long as you do so consistently:

```
int x = 7;                // definition  
extern int x;              // declaration  
extern int x;              // another declaration  
  
double sqrt(double);       // declaration  
double sqrt(double d) /* ... */ // definition  
double sqrt(double);       // another declaration of sqrt  
double sqrt(double);       // yet another declaration of sqrt  
  
int sqrt(double);          // error: inconsistent declarations of sqrt
```

Why is that last declaration an error? Because there cannot be two functions called `sqrt` taking an argument of type `double` and returning different types (`int` and `double`).

The `extern` keyword used in the second declaration of `x` simply states that this declaration of `x` isn’t a definition. It is rarely useful. We recommend that you don’t

use it, but you'll see it in other people's code, especially code that uses too many global variables (see §8.4 and §8.6.2).



Why does C++ offer both declarations and definitions? The declaration/definition distinction reflects the fundamental distinction between what we need to use something (an interface) and what we need for that something to do what it is supposed to (an implementation). For a variable, a declaration supplies the type but only the definition supplies the object (the memory). For a function, a declaration again provides the type (argument types plus return type) but only the definition supplies the function body (the executable statements). Note that function bodies are stored in memory as part of the program, so it is fair to say that function and variable definitions consume memory, whereas declarations don't.

The declaration/definition distinction allows us to separate a program into many parts that can be compiled separately. The declarations allow each part of a program to maintain a view of the rest of the program without bothering with the definitions in other parts. As all declarations (including the one definition) must be consistent, the use of names in the whole program will be consistent. We'll discuss that further in §8.3. Here, we'll just remind you of the expression parser from Chapter 6: `expression()` calls `term()` which calls `primary()` which calls `expression()`. Since every name in a C++ program has to be declared before it is used, there is no way we could just define those three functions:

```

double expression();           // just a declaration, not a definition
double primary()
{
    // ...
    expression();
    // ...
}
double term()
{
    // ...
}

```

```
    primary();
    // ...
}

double expression()
{
    // ...
    term();
    // ...
}
```

We can order those four functions any way we like; there will always be one call to a function defined below it. Somewhere, we need a “forward” declaration. Therefore, we declared `expression()` before the definition of `primary()` and all is well. Such cyclic calling patterns are very common.

Why does a name have to be declared before it is used? Couldn’t we just require the language implementation to read the program (just as we do) and find the definition to see how a function must be called? We could, but that would lead to “interesting” technical problems, so we decided against that. The C++ definition requires declaration before use (except for class members; see §9.4.4). After all, this is already the convention for ordinary (non-program) writing: when you read a textbook, you expect the author to define terminology before using it; otherwise, you have to guess or go to the index all the time. The “declaration before use” rule simplifies reading for both humans and compilers. In a program, there is a second reason that “declare before use” is important. In a program of thousands of lines (maybe hundreds of thousands of lines), most of the functions we want to call will be defined “elsewhere.” That “elsewhere” is often a place we don’t really want to know about. Having to know the declarations only of what we use saves us (and the compiler) from looking through huge amounts of program text.

8.2.1 Kinds of declarations

There are many kinds of entities that a programmer can define in C++. The most interesting are

- Variables
- Constants
- Functions (see §8.5)
- Namespaces (see §8.7)
- Types (classes and enumerations; see Chapter 9)
- Templates (see Chapter 19)

8.2.2 Variable and constant declarations

The declaration of a variable or a constant specifies a name, a type, and optionally an initializer. For example:

```
int a;                                // no initializer
double d = 7;                      // initializer using the = syntax
vector<int> vi(10);                // initializer using the () syntax
vector<int> vi2 {1,2,3,4};          // initializer using the {} syntax
```

You can find the complete grammar in the ISO C++ standard.

Constants have the same declaration syntax as variables. They differ in having **const** as part of their type and requiring an initializer:

```
const int x = 7;                    // initializer using the = syntax
const int x2 {9};                  // initializer using the {} syntax
const int y;                      // error: no initializer
```

The reason for requiring an initializer for a **const** is obvious: how could a **const** be a constant if it didn't have a value? It is almost always a good idea to initialize variables also; an uninitialized variable is a recipe for obscure bugs. For example:

```
void f(int z)
{
    int x;                      // uninitialized
    // ... no assignment to x here ...
    x = 7;                      // give x a value
    // ...
}
```

This looks innocent enough, but what if the first **...** included a use of **x**? For example:

```
void f(int z)
{
    int x;                      // uninitialized
    // ... no assignment to x here ...
    if (z>x) {
        // ...
    }
    // ...
```

```
x = 7;           // give x a value
// ...
}
```

Because `x` is uninitialized, executing `z>x` would be undefined behavior. The comparison `z>x` could give different results on different machines and different results in different runs of the program on the same machine. In principle, `z>x` might cause the program to terminate with a hardware error, but most often that doesn't happen. Instead we get unpredictable results.

Naturally, we wouldn't do something like that deliberately, but if we don't consistently initialize variables it will eventually happen by mistake. Remember, most "silly mistakes" (such as using an uninitialized variable before it has been assigned to) happen when you are busy or tired. Compilers try to warn, but in complicated code – where such errors are most likely to occur – compilers are not smart enough to catch all such errors. There are people who are not in the habit of initializing their variables, often because they learned to program in languages that didn't allow or encourage consistent initialization; so you'll see examples in other people's code. Please just don't add to the problem by forgetting to initialize the variables you define yourself.

We have a preference for the `{ }` initializer syntax. It is the most general and it most explicitly says "initializer." We tend to use it except for very simple initializations, where we sometimes use `=` out of old habits, and `()` for specifying the number of elements of a `vector` (see §17.4.4).

8.2.3 Default initialization

You might have noticed that we often don't provide an initializer for `strings`, `vectors`, etc. For example:

```
vector<string> v;
string s;
while (cin>>s) v.push_back(s);
```

This is not an exception to the rule that variables must be initialized before use. What is going on here is that `string` and `vector` are defined so that variables of those types are initialized with a default value whenever we don't supply one explicitly. Thus, `v` is empty (it has no elements) and `s` is the empty string ("") before we reach the loop. The mechanism for guaranteeing default initialization is called a *default constructor*; see §9.7.3.

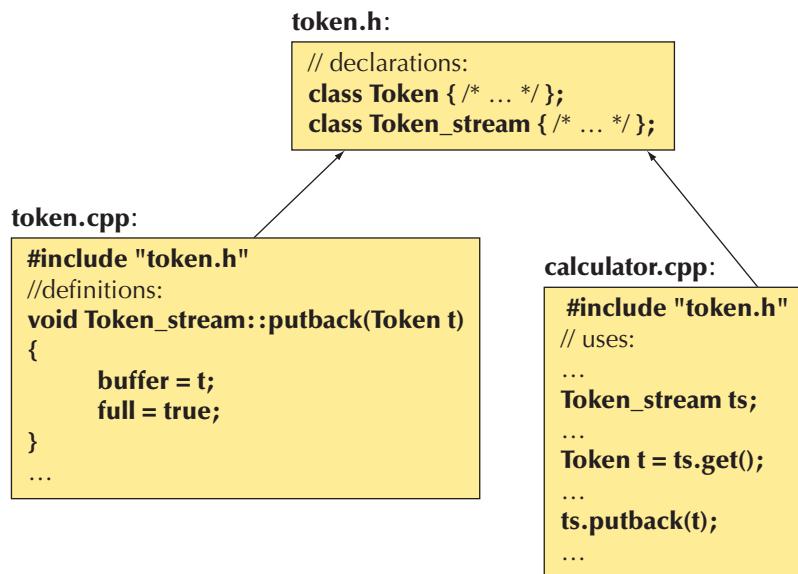
Unfortunately, the language doesn't allow us to make such guarantees for built-in types. A global variable (§8.4) is default initialized to 0, but you should minimize the use of global values. The most useful variables, local variables and

class members, are uninitialized unless you provide an initializer (or a default constructor). You have been warned!

8.3 Header files

How do we manage our declarations and definitions? After all, they have to be consistent, and in real-world programs there can be tens of thousands of declarations; programs with hundreds of thousands of declarations are not rare. Typically, when we write a program, most of the definitions we use are not written by us. For example, the implementations of `cout` and `sqrt()` were written by someone else many years ago. We just use them.

The key to managing declarations of facilities defined “elsewhere” in C++ is the header. Basically, a *header* is a collection of declarations, typically defined in a file, so a header is also called a *header file*. Such headers are then **#included** in our source files. For example, we might decide to improve the organization of the source code for our calculator (Chapters 6 and 7) by separating out the token management. We could define a header file `token.h` containing declarations needed to use `Token` and `Token_stream`:



The declarations of `Token` and `Token_stream` are in the header `token.h`. Their definitions are in `token.cpp`. The `.h` suffix is the most common for C++ headers, and the `.cpp` suffix is the most common for C++ source files. Actually, the C++ language doesn't care about file suffixes, but some compilers and most program

development environments insist, so please use this convention for your source code.

In principle, `#include "file.h"` simply copies the declarations from `file.h` into your file at the point of the `#include`. For example, we could write a header `f.h`:

```
// f.h
int f(int);
```

and include it in our file `user.cpp`:

```
// user.cpp
#include "f.h"
int g(int i)
{
    return f(i);
}
```

When compiling `user.cpp` the compiler would do the `#include` and compile

```
int f(int);
int g(int i)
{
    return f(i);
}
```

Since `#includes` logically happen before anything else a compiler does, handling `#includes` is part of what is called *preprocessing* (§A.17).

To ease consistency checking, we `#include` a header both in source files that use its declarations and in source files that provide definitions for those declarations. That way, the compiler catches errors as soon as possible. For example, imagine that the implementer of `Token_stream::putback()` made mistakes:

```
Token Token_stream::putback(Token t)
{
    buffer.push_back(t);
    return t;
}
```

This looks innocent enough. Fortunately, the compiler catches the mistakes because it sees the (`#included`) declaration of `Token_stream::putback()`. Comparing that declaration with our definition, the compiler finds that `putback()` should not return a `Token` and that `buffer` is a `Token`, rather than a `vector<Token>`, so we

can't use `push_back()`. Such mistakes occur when we work on our code to improve it, but don't quite get a change consistent throughout a program.

Similarly, consider these mistakes:

```
Token t = ts.gett();      // error: no member gett
// . . .
ts.putback();           // error: argument missing
```

The compiler would immediately give errors; the header `token.h` gives it all the information it needs for checking.

Our `std_lib_facilities.h` header contains declarations for the standard library facilities we use, such as `cout`, `vector`, and `sqrt()`, together with a couple of simple utility functions, such as `error()`, that are not part of the standard library. In §12.8 we show how to use the standard library headers directly.

A header will typically be included in many source files. That means that a header should only contain declarations that can be duplicated in several files (such as function declarations, class definitions, and definitions of numeric constants).

8.4 Scope

A *scope* is a region of program text. A name is declared in a scope and is valid (is “in scope”) from the point of its declaration until the end of the scope in which it was declared. For example:

```
void f()
{
    g();           // error: g() isn't (yet) in scope
}

void g()
{
    f();           // OK: f() is in scope
}

void h()
{
    int x = y;    // error: y isn't (yet) in scope
    int y = x;    // OK: x is in scope
    g();           // OK: g() is in scope
}
```

Names in a scope can be seen from within scopes nested within it. For example, the call of `f()` is within the scope of `g()` which is “nested” in the global scope. The global scope is the scope that’s not nested in any other. The rule that a name must be declared before it can be used still holds, so `f()` cannot call `g()`.

There are several kinds of scopes that we use to control where our names can be used:

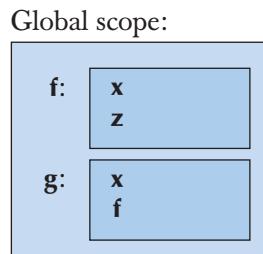
- The *global scope*: the area of text outside any other scope
- A *namespace scope*: a named scope nested in the global scope or in another namespace; see §8.7
- A *class scope*: the area of text within a class; see §9.2
- A *local scope*: between `{...}` braces of a block or in a function argument list
- A *statement scope*: e.g., in a `for`-statement

The main purpose of a scope is to keep names local, so that they won’t interfere with names declared elsewhere. For example:

```
void f(int x)           // f is global; x is local to f
{
    int z = x+7;       // z is local
}

int g(int x)           // g is global; x is local to g
{
    int f = x+2;       // f is local
    return 2*f;
}
```

Or graphically:



Here `f()`’s `x` is different from `g()`’s `x`. They don’t “clash” because they are not in the same scope: `f()`’s `x` is local to `f` and `g()`’s `x` is local to `g`. Two incompatible

declarations in the same scope are often referred to as a *clash*. Similarly, the **f** defined and used within **g()** is (obviously) not the function **f()**.

Here is a logically equivalent but more realistic example of the use of local scope:

```
int max(int a, int b)      // max is global; a and b are local
{
    return (a>=b) ? a : b;
}

int abs(int a)            // not max()'s a
{
    return (a<0) ? -a : a;
}
```

You find **max()** and **abs()** in the standard library, so you don't have to write them yourself. The **?:** construct is called an *arithmetic if* or a *conditional expression*. The value of **(a>=b)?a:b** is **a** if **a>=b** and **b** otherwise. A conditional expression saves us from writing long-winded code like this:

```
int max(int a, int b)      // max is global; a and b are local
{
    int m;                  // m is local
    if (a>=b)
        m = a;
    else
        m = b;
    return m;
}
```

So, with the notable exception of the global scope, a scope keeps names local. For most purposes, locality is good, so keep names as local as possible. When I declare my variables, functions, etc. within functions, classes, namespaces, etc., they won't interfere with yours. Remember: Real programs have *many* thousands of named entities. To keep such programs manageable, most names have to be local.

Here is a larger technical example illustrating how names go out of scope at the end of statements and blocks (including function bodies):

```
// no r, i, or v here
class My_vector {
    vector<int> v;    // v is in class scope
```

```

public:
    int largest()
    {
        int r = 0;           // r is local (smallest nonnegative int)
        for (int i = 0; i < v.size(); ++i)
            r = max(r,abs(v[i]));   // i is in the for's statement scope
        // no i here
        return r;
    }
    // no r here
};

// no v here

int x;           // global variable — avoid those where you can
int y;

int f()
{
    int x;           // local variable, hides the global x
    x = 7;          // the local x
    {
        int x = y; // local x initialized by global y, hides the previous local x
        ++x;          // the x from the previous line
    }
    ++x;          // the x from the first line of f()
    return x;
}

```

Whenever you can, avoid such complicated nesting and hiding. Remember: “Keep it simple!”

The larger the scope of a name is, the longer and more descriptive its name should be: **x**, **y**, and **f** are horrible as global names. The main reason that you don’t want global variables in your program is that it is hard to know which functions modify them. In large programs, it is basically impossible to know which functions modify a global variable. Imagine that you are trying to debug a program and you find that a global variable has an unexpected value. Who gave it that value? Why? What functions write to that value? How would you know? The function that wrote a bad value to that variable may be in a source file you have never seen! A good program will have only very few (say, one or two), if any, global variables. For example, the calculator in Chapters 6 and 7 had two global variables: the token stream, **ts**, and the symbol table, **names**.

Note that most C++ constructs that define scopes nest:

- Functions within classes: member functions (see §9.4.2)

```
class C {
public:
    void f();
    void g()    // a member function can be defined within its class
    {
        // ...
    }
// ...
};

void C::f()      // a member definition can be outside its class
{
    // ...
}
```

This is the most common and useful case.

- Classes within classes: member classes (also called nested classes)

```
class C {
public:
    struct M {
        // ...
    };
// ...
};
```

This tends to be useful only in complicated classes; remember that the ideal is to keep classes small and simple.

- Classes within functions: local classes

```
void f()
{
    class L {
        // ...
    };
// ...
}
```

Avoid this; if you feel the need for a local class, your function is probably far too long.



- Functions within functions: local functions (also called nested functions)

```
void f()
{
    void g()      // illegal
    {
        // ...
    }
    // ...
}
```

This is not legal in C++; don't do it. The compiler will reject it.

- Blocks within functions and other blocks: nested blocks

```
void f(int x, int y)
{
    if (x>y) {
        // ...
    }
    else {
        // ...
    {
        // ...
    }
    // ...
}
}
```

Nested blocks are unavoidable, but be suspicious of complicated nesting: it can easily hide errors.

C++ also provides a language feature, **namespace**, exclusively for expressing scoping; see §8.7.

Note our consistent indentation to indicate nesting. Without consistent indentation, nested constructs become unreadable. For example:

```
// dangerously ugly code
struct X {
    void f(int x) {
        struct Y {
            int f() { return 1; } int m; ;
            int m;
            m=x; Y m2;
        }
    }
}
```



```

return f(m2.f()); }
int m; void g(int m) {
if (m) f(m+2); else {
g(m+2); }}
X() {} void m3() {
}

void main() {
X a; a.f(2);}
};

```

Hard-to-read code usually hides bugs. When you use an IDE, it tries to automatically make your code properly indented (according to some definition of “properly”), and there exist “code beautifiers” that will reformat a source code file for you (often offering you a choice of formats). However, the ultimate responsibility for your code being readable rests with you.

8.5 Function call and return

 Functions are the way we represent actions and computations. Whenever we want to do something that is worthy of a name, we write a function. The C++ language gives us operators (such as `+` and `*`) with which we can produce new values from operands in expressions, and statements (such as `for` and `if`) with which we can control the order of execution. To organize code made out of these primitives, we have functions.

To do its job, a function usually needs arguments, and many functions return a result. This section focuses on how arguments are specified and passed.

8.5.1 Declaring arguments and return type

Functions are what we use in C++ to name and represent computations and actions. A function declaration consists of a return type followed by the name of the function followed by a list of formal arguments in parentheses. For example:

```

double fct(int a, double d);           // declaration of fct (no body)
double fct(int a, double d) { return a*d; }    // definition of fct

```

A definition contains the function body (the statements to be executed by a call), whereas a declaration that isn’t a definition just has a semicolon. Formal arguments are often called *parameters*. If you don’t want a function to take arguments, just leave out the formal arguments. For example:

```

int current_power();           // current_power doesn't take an argument

```

If you don't want to return a value from a function, give **void** as its return type. For example:

```
void increase_power(int level);      // increase_power doesn't return a value
```

Here, **void** means “doesn't return a value” or “return nothing.”

You can name a parameter or not as it suits you in both declarations and definitions. For example:

```
// search for s in vs;
// vs[hint] might be a good place to start the search
// return the index of a match; -1 indicates "not found"
int my_find(vector<string> vs, string s, int hint); // naming arguments

int my_find(vector<string>, string, int);           // not naming arguments
```

In declarations, formal argument names are not logically necessary, just very useful for writing good comments. From a compiler's point of view, the second declaration of **my_find()** is just as good as the first: it has all the information necessary to call **my_find()**.

Usually, we name all the arguments in the definition. For example:

```
int my_find(vector<string> vs, string s, int hint)
// search for s in vs starting at hint
{
    if (hint<0 || vs.size()<=hint) hint = 0;
    for (int i = hint; i<vs.size(); ++i) // search starting from hint
        if (vs[i]==s) return i;
    if (0<hint)                                // if we didn't find s search before hint
        for (int i = 0; i<hint; ++i)
            if (vs[i]==s) return i;
    }
    return -1;
}
```

The **hint** complicates the code quite a bit, but the **hint** was provided under the assumption that users could use it to good effect by knowing roughly where in the **vector** a **string** will be found. However, imagine that we had used **my_find()** for a while and then discovered that callers rarely used **hint** well, so that it actually hurt performance. Now we don't need **hint** anymore, but there is lots of code “out there” that calls **my_find()** with a **hint**. We don't want to rewrite that code (or can't because it is someone else's code), so we don't want to change the

declaration(s) of **my_find()**. Instead, we just don't use the last argument. Since we don't use it we can leave it unnamed:

```
int my_find(vector<string> vs, string s, int)      // 3rd argument unused
{
    for (int i = 0; i<vs.size(); ++i)
        if (vs[i]==s) return i;
    return -1;
}
```

You can find the complete grammar for function definitions in the ISO C++ standard.

8.5.2 Returning a value

We return a value from a function using a **return**-statement:

```
T f()      // f() returns a T
{
    V v;
    ...
    return v;
}

T x = f();
```

Here, the value returned is exactly the value we would have gotten by initializing a variable of type **T** with a value of type **V**:

```
V v;
...
T t(v); // initialize t with v
```

That is, value return is a form of initialization.

A function declared to return a value must return a value. In particular, it is an error to “fall through the end of the function”:

```
double my_abs(int x)      // warning: buggy code
{
    if (x < 0)
        return -x;
    else if (x > 0)
        return x;
}      // error: no value returned if x is 0
```

Actually, the compiler probably won't notice that we "forgot" the case `x==0`. In principle it could, but few compilers are that smart. For complicated functions, it can be impossible for a compiler to know whether or not you return a value, so be careful. Here, "being careful" means to make really sure that you have a `return`-statement or an `error()` for every possible way out of the function.

For historical reasons, `main()` is a special case. Falling through the bottom of `main()` is equivalent to returning the value `0`, meaning "successful completion" of the program.

In a function that does not return a value, we can use `return` without a value to cause a return from the function. For example:

```
void print_until_s(vector<string> v, string quit)
{
    for(int s : v) {
        if (s==quit) return;
        cout << s << '\n';
    }
}
```

As you can see, it is acceptable to "drop through the bottom" of a `void` function. This is equivalent to a `return;`.

8.5.3 Pass-by-value

The simplest way of passing an argument to a function is to give the function a copy of the value you use as the argument. An argument of a function `f()` is a local variable in `f()` that's initialized each time `f()` is called. For example:

```
// pass-by-value (give the function a copy of the value passed)
int f(int x)
{
    x = x+1;           // give the local x a new value
    return x;
}

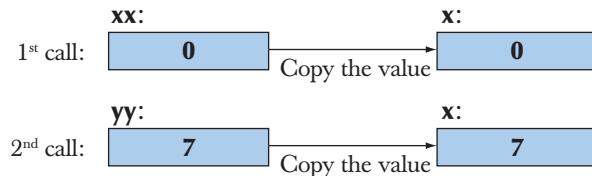
int main()
{
    int xx = 0;
    cout << f(xx) << '\n';    // write: 1
    cout << xx << '\n';      // write: 0; f() doesn't change xx
}
```

```

int yy = 7;
cout << f(yy) << '\n';           // write: 8
cout << yy << '\n';             // write: 7; f() doesn't change yy
}

```

Since a copy is passed, the `x=x+1` in `f()` does not change the values `xx` and `yy` passed in the two calls. We can illustrate a pass-by-value argument passing like this:



Pass-by-value is pretty straightforward and its cost is the cost of copying the value.

8.5.4 Pass-by-const-reference

Pass-by-value is simple, straightforward, and efficient when we pass small values, such as an `int`, a `double`, or a `Token` (§6.3.2). But what if a value is large, such as an image (often, several million bits), a large table of values (say, thousands of integers), or a long string (say, hundreds of characters)? Then, copying can be costly. We should not be obsessed by cost, but doing unnecessary work can be embarrassing because it is an indication that we didn't directly express our idea of what we wanted. For example, we could write a function to print out a `vector` of floating-point numbers like this:

```

void print(vector<double> v)           // pass-by-value; appropriate?
{
    cout << "{ ";
    for (int i = 0; i < v.size(); ++i) {
        cout << v[i];
        if (i != v.size() - 1) cout << ", ";
    }
    cout << " }\n";
}

```

We could use this `print()` for `vectors` of all sizes. For example:

```

void f(int x)
{
    vector<double> vd1(10);           // small vector
    vector<double> vd2(1000000);      // large vector
}

```

```

vector<double> vd3(x);           // vector of some unknown size
// . . . fill vd1, vd2, vd3 with values . .
print(vd1);
print(vd2);
print(vd3);
}

```

This code works, but the first call of **print()** has to copy ten **doubles** (probably 80 bytes), the second call has to copy a million **doubles** (probably 8 megabytes), and we don't know how much the third call has to copy. The question we must ask ourselves here is: "Why are we copying anything at all?" We just wanted to print the **vectors**, not to make copies of their elements. Obviously, there has to be a way for us to pass a variable to a function without copying it. As an analogy, if you were given the task to make a list of books in a library, the librarians wouldn't ship you a copy of the library building and all its contents; they would send you the address of the library, so that you could go and look at the books. So, we need a way of giving our **print()** function "the address" of the **vector** to **print()** rather than the copy of the **vector**. Such an "address" is called a *reference* and is used like this:

```

void print(const vector<double>& v) // pass-by-const-reference
{
    cout << "{ ";
    for (int i = 0; i < v.size(); ++i) {
        cout << v[i];
        if (i != v.size() - 1) cout << ", ";
    }
    cout << " }\n";
}

```

The **&** means "reference" and the **const** is there to stop **print()** modifying its argument by accident. Apart from the change to the argument declaration, all is the same as before; the only change is that instead of operating on a copy, **print()** now refers back to the argument through the reference. Note the phrase "refer back"; such arguments are called references because they "refer" to objects defined elsewhere. We can call this **print()** exactly as before:

```

void f(int x)
{
    vector<double> vd1(10);           // small vector
    vector<double> vd2(1000000);      // large vector
    vector<double> vd3(x);           // vector of some unknown size
// . . . fill vd1, vd2, vd3 with values . .

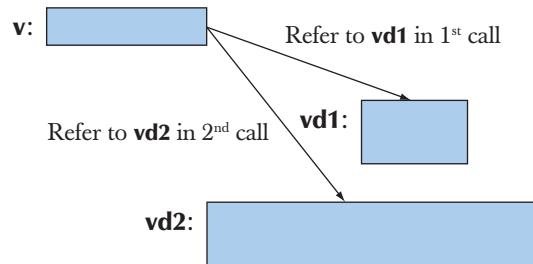
```

```

    print(vd1);
    print(vd2);
    print(vd3);
}

```

We can illustrate that graphically:



A **const** reference has the useful property that we can't accidentally modify the object passed. For example, if we made a silly error and tried to assign to an element from within **print()**, the compiler would catch it:

```

void print(const vector<double>& v)      // pass-by-const-reference
{
    // ...
    v[i] = 7;                                // error: v is a const (is not mutable)
    // ...
}

```

Pass-by-**const**-reference is a useful and popular mechanism. Consider again the **my_find()** function (§8.5.1) that searches for a **string** in a **vector** of **strings**. Pass-by-value could be unnecessarily costly:

```
int my_find(vector<string> vs, string s); // pass-by-value: copy
```

If the **vector** contained thousands of **strings**, you might notice the time spent even on a fast computer. So, we could improve **my_find()** by making it take its arguments by **const** reference:

```
// pass-by-const-reference: no copy, read-only access
int my_find(const vector<string>& vs, const string& s);
```

8.5.5 Pass-by-reference

But what if we did want a function to modify its arguments? Sometimes, that's a perfectly reasonable thing to wish for. For example, we might want an `init()` function that assigns values to `vector` elements:

```
void init(vector<double>& v)           // pass-by-reference
{
    for (int i = 0; i < v.size(); ++i) v[i] = i;
}

void g(int x)
{
    vector<double> vd1(10);           // small vector
    vector<double> vd2(1000000);      // large vector
    vector<double> vd3(x);           // vector of some unknown size

    init(vd1);
    init(vd2);
    init(vd3);
}
```

Here, we wanted `init()` to modify the argument vector, so we did not copy (did not use pass-by-value) or declare the reference `const` (did not use pass-by-`const`-reference) but simply passed a “plain reference” to the `vector`.

Let us consider references from a more technical point of view. A reference is a construct that allows a user to declare a new name for an object. For example, `int&` is a reference to an `int`, so we can write

```
int i = 7;

int& r = i;    // r is a reference to i
r = 9;         // i becomes 9
i = 10;
cout << r << ' ' << i << '\n'; // write: 10 10
```



That is, any use of `r` is really a use of `i`.

References can be useful as shorthand. For example, we might have a

```
vector<vector<double>> v; // vector of vector of double
```

and we need to refer to some element `v[f(x)][g(y)]` several times. Clearly, `v[f(x)]` [`g(y)`] is a complicated expression that we don't want to repeat more often than we have to. If we just need its value, we could write

```
double val = v[f(x)][g(y)];           // val is the value of v[f(x)][g(y)]
```

and use `val` repeatedly. But what if we need to both read from `v[f(x)][g(y)]` and write to `v[f(x)][g(y)]`? Then, a reference comes in handy:

```
double& var = v[f(x)][g(y)];           // var is a reference to v[f(x)][g(y)]
```

Now we can read and write `v[f(x)][g(y)]` through `var`. For example:

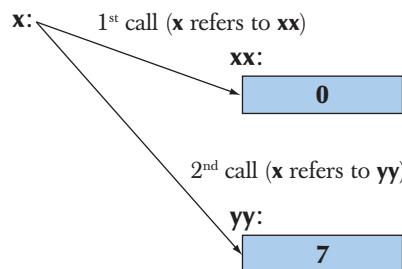
```
var = var/2+sqrt(var);
```

This key property of references, that a reference can be a convenient shorthand for some object, is what makes them useful as arguments. For example:

```
// pass-by-reference (let the function refer back to the variable passed)
int f(int& x)
{
    x = x+1;
    return x;
}
int main()
{
    int xx = 0;
    cout << f(xx) << '\n';           // write: 1
    cout << xx << '\n';           // write: 1; f() changed the value of xx

    int yy = 7;
    cout << f(yy) << '\n';           // write: 8
    cout << yy << '\n';           // write: 8; f() changed the value of yy
}
```

We can illustrate a pass-by-reference argument passing like this:





Compare this to the similar example in §8.5.3.

Pass-by-reference is clearly a very powerful mechanism: we can have a function operate directly on any object to which we pass a reference. For example, swapping two values is an immensely important operation in many algorithms, such as sorting. Using references, we can write a function that swaps **doubles** like this:

```
void swap(double& d1, double& d2)
{
    double temp = d1;           // copy d1's value to temp
    d1 = d2;                   // copy d2's value to d1
    d2 = temp;                 // copy d1's old value to d2
}

int main()
{
    double x = 1;
    double y = 2;
    cout << "x == " << x << " y == " << y << '\n'; // write: x==1 y==2
    swap(x,y);
    cout << "x == " << x << " y == " << y << '\n'; // write: x==2 y==1
}
```

The standard library provides a **swap()** for every type that you can copy, so you don't have to write **swap()** yourself for each type.

8.5.6 Pass-by-value vs. pass-by-reference

When should you use pass-by-value, pass-by-reference, and pass-by-**const**-reference? Consider first a technical example:

```
void f(int a, int& r, const int& cr)
{
    ++a;           // change the local a
    ++r;           // change the object referred to by r
    ++cr;          // error: cr is const
}
```

If you want to change the value of the object passed, you must use a non-**const** reference: pass-by-value gives you a copy and pass-by-**const**-reference prevents you from changing the value of the object passed. So we can try

```
void g(int a, int& r, const int& cr)
{
    ++a;          // change the local a
    ++r;          // change the object referred to by r
    int x = cr;  // read the object referred to by cr
}

int main()
{
    int x = 0;
    int y = 0;
    int z = 0;

    g(x,y,z);      // x==0; y==1; z==0
    g(1,2,3);      // error: reference argument r needs a variable to refer to
    g(1,y,3);      // OK: since cr is const we can pass a literal
}
```

So, if you want to change the value of an object passed by reference, you have to pass an object. Technically, the integer literal **2** is just a value (an rvalue), rather than an object holding a value. What you need for **g()**'s argument **r** is an lvalue, that is, something that could appear on the left-hand side of an assignment.

Note that a **const** reference doesn't need an lvalue. It can perform conversions exactly as initialization or pass-by-value. Basically, what happens in that last call, **g(1,y,3)**, is that the compiler sets aside an **int** for **g()**'s argument **cr** to refer to:

```
g(1,y,3);  // means: int __compiler_generated = 3; g(1,y,__compiler_generated)
```

Such a compiler-generated object is called a *temporary object* or just a *temporary*.

Our rule of thumb is:

1. Use pass-by-value to pass very small objects.
2. Use pass-by-**const**-reference to pass large objects that you don't need to modify.
3. Return a result rather than modifying an object through a reference argument.
4. Use pass-by-reference only when you have to.



These rules lead to the simplest, least error-prone, and most efficient code. By “very small” we mean one or two **ints**, one or two **doubles**, or something like that. If we see an argument passed by non-**const** reference, we must assume that the called function will modify that argument.

That third rule reflects that you have a choice when you want to use a function to change the value of a variable. Consider:

```
int incr1(int a) { return a+1; }      // return the new value as the result
void incr2(int& a) { ++a; }          // modify object passed as reference

int x = 7;                         // pretty obvious
x = incr1(x);                     // pretty obscure
incr2(x);
```

Why do we ever use non-**const**-reference arguments? Occasionally, they are 

- For manipulating containers (e.g., **vector**) and other large objects
- For functions that change several objects (we can have only one return value)

For example:

```
void larger(vector<int>& v1, vector<int>& v2)
    // make each element in v1 the larger of the corresponding
    // elements in v1 and v2;
    // similarly, make each element of v2 the smaller
{
    if (v1.size()!=v2.size()) error("larger(): different sizes");
    for (int i=0; i<v1.size(); ++i)
        if (v1[i]<v2[i])
            swap(v1[i],v2[i]);
}
void f()
{
    vector<int> vx;
    vector<int> vy;
    // read vx and vy from input
    larger(vx,vy);
    // ...
}
```

Using pass-by-reference arguments is the only reasonable choice for a function like **larger()**.

It is usually best to avoid functions that modify several objects. In theory, there are always alternatives, such as returning a class object holding several values. However, there are a lot of programs “out there” expressed in terms of functions that modify one or more arguments, so you are likely to encounter them. For example, in Fortran – the major programming language used for numerical calculation for about 50 years – all arguments are traditionally passed by reference. Many numeric programmers copy Fortran designs and call functions written in Fortran. Such code often uses pass-by-reference or pass-by-**const**-reference.

If we use a reference simply to avoid copying, we use a **const** reference. Consequently, when we see a non-**const**-reference argument, we assume that the function changes the value of its argument; that is, when we see a pass-by-non-**const**-reference we assume that not only can that function modify the argument passed, but it will, so that we have to look extra carefully at the call to make sure that it does what we expect it to.

8.5.7 Argument checking and conversion

Passing an argument is the initialization of the function’s formal argument with the actual argument specified in the call. Consider:

```
void f(T x);
f(y);
T x = y;           // initialize x with y (see §8.2.2)
```

The call **f(y)** is legal whenever the initialization **T x=y;** is, and when it is legal both **x**s get the same value. For example:

```
void f(double x);
void g(int y)
{
    f(y);
    double x = y;   // initialize x with y (see §8.2.2)
}
```

Note that to initialize **x** with **y**, we have to convert an **int** to a **double**. The same happens in the call of **f()**. The **double** value received by **f()** is the same as the one stored in **x**.

Conversions are often useful, but occasionally they give surprising results (see §3.9.2). Consequently, we have to be careful with them. Passing a **double** as an argument to a function that requires an **int** is rarely a good idea:

```
void ff(int x);

void gg(double y)
{
    ff(y);           // how would you know if this makes sense?
    int x = y;       // how would you know if this makes sense?

}
```

If you really mean to truncate a **double** value to an **int**, say so explicitly:

```
void ggg(double x)
{
    int x1 = x;           // truncate x
    int x2 = int(x);
    int x3 = static_cast<int>(x); // very explicit conversion (§17.8)

    ff(x1);
    ff(x2);
    ff(x3);

    ff(x);           // truncate x
    ff(int(x));
    ff(static_cast<int>(x)); // very explicit conversion (§17.8)

}
```

That way, the next programmer to look at this code can see that you thought about the problem.

8.5.8 Function call implementation

But how does a computer really do a function call? The **expression()**, **term()**, and **primary()** functions from Chapters 6 and 7 are perfect for illustrating this except for one detail: they don't take any arguments, so we can't use them to explain how arguments are passed. But wait! They *must* take some input; if they didn't, they couldn't do anything useful. They do take an implicit argument: they use a **Token_stream** called **ts** to get their input; **ts** is a global variable. That's a bit sneaky. We can improve these functions by letting them take a **Token_stream&** argument. Here they are with a **Token_stream&** parameter added and everything that doesn't concern function call implementation removed.

First, `expression()` is completely straightforward; it has one argument (`ts`) and two local variables (`left` and `t`):

```
double expression(Token_stream& ts)
{
    double left = term(ts);
    Token t = ts.get();
    // ...
}
```

Second, `term()` is much like `expression()`, except that it has an additional local variable (`d`) that it uses to hold the result of a divisor for `'/'`:

```
double term(Token_stream& ts)
{
    double left = primary(ts);
    Token t = ts.get();
    // ...
    case '/':
    {
        double d = primary(ts);
        // ...
    }
    // ...
}
```

Third, `primary()` is much like `term()` except that it doesn't have a local variable `left`:

```
double primary(Token_stream& ts)
{
    Token t = ts.get();
    switch (t.kind) {
        case '(':
            {   double d = expression(ts);
                // ...
            }
            // ...
        }
    }
}
```

Now they don't use any “sneaky global variables” and are perfect for our illustration: they have an argument, they have local variables, and they call each other.

You may want to take the opportunity to refresh your memory of what the complete **expression()**, **term()**, and **primary()** look like, but the salient features as far as function call is concerned are presented here.

When a function is called, the language implementation sets aside a data structure containing a copy of all its parameters and local variables. For example, when **expression()** is first called, the compiler ensures that a structure like this is created:

Call of expression() :
ts
left
t
Implementation stuff

The “implementation stuff” varies from implementation to implementation, but that’s basically the information that the function needs to return to its caller and to return a value to its caller. Such a data structure is called a *function activation record*, and each function has its own detailed layout of its activation record. Note that from the implementation’s point of view, a parameter is just another local variable.

So far, so good, and now **expression()** calls **term()**, so the compiler ensures that an activation record for this call of **term()** is generated:

Call of expression() :
ts
left
t
Implementation stuff

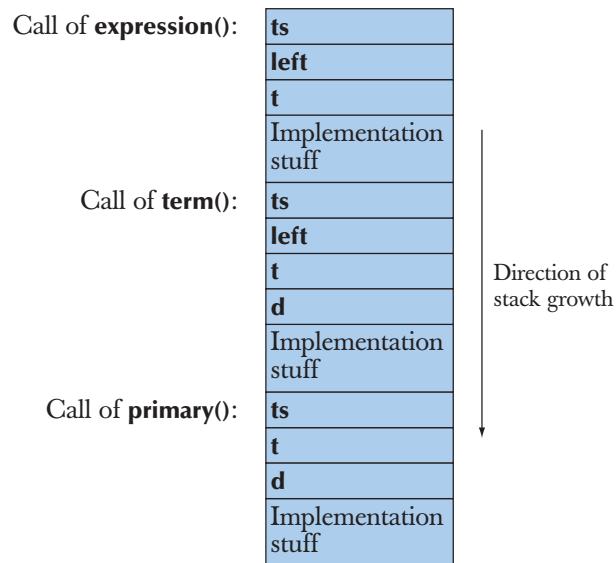
Call of term() :
ts
left
t
d
Implementation stuff

↓

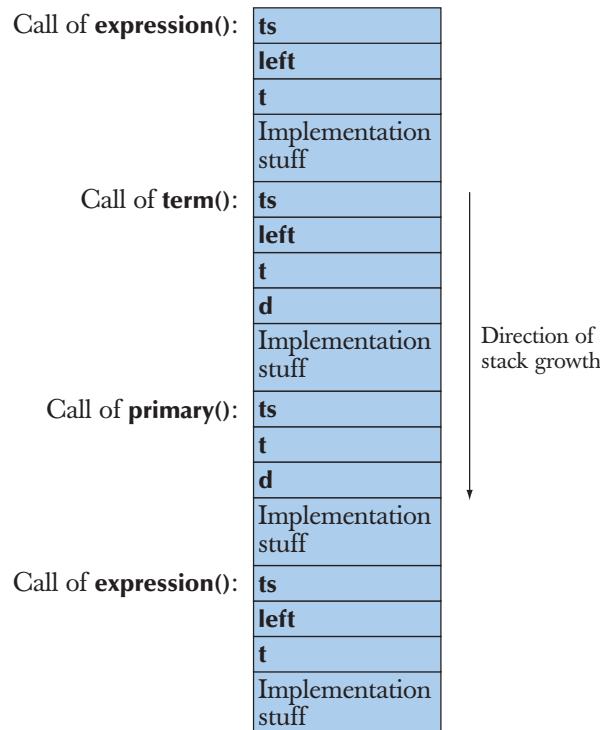
Direction of stack growth

Note that **term()** has an extra variable **d** that needs to be stored, so we set aside space for that in the call even though the code may never get around to using it. That’s OK. For reasonable functions (such as every function we directly or indirectly use in this book), the run-time cost of laying down a function activation record doesn’t depend on how big it is. The local variable **d** will be initialized only if we execute its **case '/'**.

Now **term()** calls **primary()** and we get



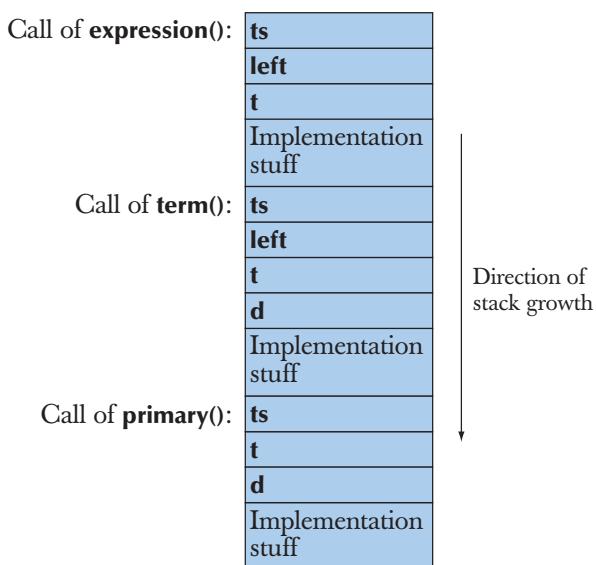
This is starting to get a bit repetitive, but now **primary()** calls **expression()**:



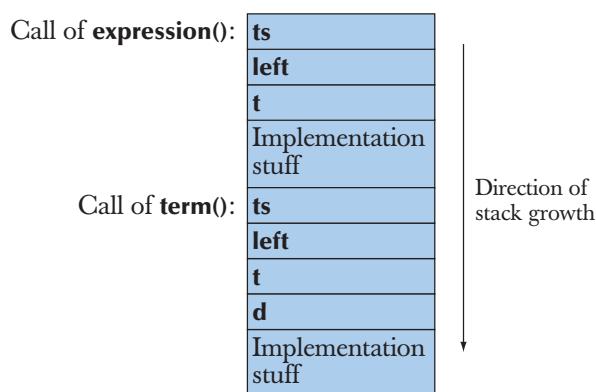


So this call of **expression()** gets its own activation record, different from the first call of **expression()**. That's good or else we'd be in a terrible mess, since **left** and **t** will be different in the two calls. A function that directly or (as here) indirectly calls itself is called *recursive*. As you see, recursive functions follow naturally from the implementation technique we use for function call and return (and vice versa).

So, each time we call a function the *stack of activation records*, usually just called the *stack*, grows by one record. Conversely, when the function returns, its record is no longer used. For example, when that last call of **expression()** returns to **primary()**, the stack will revert to this:



And when that call of **primary()** returns to **term()**, we get back to



And so on. The stack, also called the *call stack*, is a data structure that grows and shrinks at one end according to the rule “Last in, first out.”

Please remember that the details of how a call stack is implemented and used vary from C++ implementation to C++ implementation, but the basics are as outlined here. Do you need to know how function calls are implemented to use them? Of course not; you have done well enough before this implementation subsection, but many programmers like to know and many use phrases like “activation record” and “call stack,” so it’s better to know what they mean.

8.5.9 `constexpr` functions

A function represents a calculation, and sometimes we want to do a calculation at compile time. The reason to want a calculation to be evaluated by the compiler is usually to avoid having the same calculation done millions of times at run time. We use functions to make our calculations comprehensible, so naturally we sometimes want to use a function in a constant expression. We convey our intent to have a function evaluated by the compiler by declaring the function `constexpr`. A `constexpr` function can be evaluated by the compiler if it is given constant expressions as arguments. For example:

```
constexpr double xscale = 10;    // scaling factors
constexpr double yscale = 0.8;

constexpr Point scale(Point p) { return {xscale*p.x,yscale*p.y}; };
```

Assume that `Point` is a simple `struct` with members `x` and `y` representing 2D coordinates. Now, when we give `scale()` a `Point` argument, it returns a `Point` with coordinates scaled according to the factors `xscale` and `yscale`. For example:

```
void user(Point p1)
{
    Point p2 {10,10};

    Point p3 = scale(p1);    // OK: p3 == {100,8}; run-time evaluation is fine
    Point p4 = scale(p2);    // p4 == {100,8}

    constexpr Point p5 = scale(p1); // error: scale (p1) is not a constant
                                  // expression
    constexpr Point p6 = scale(p2); // p6 == {100,8}

    // ...
}
```

A **constexpr** function behaves just like an ordinary function until you use it where a constant is needed. Then, it is calculated at compile time provided its arguments are constant expressions (e.g., **p2**) and gives an error if they are not (e.g., **p1**). To enable that, a **constexpr** function must be so simple that the compiler (every standard-conforming compiler) can evaluate it. In C++11, that means that a **constexpr** function must have a body consisting of a single **return**-statement (like **scale()**); in C++14, we can also write simple loops. A **constexpr** function may not have side effects; that is, it may not change the value of variables outside its own body, except those it is assigned to or uses to initialize.

Here is an example of a function that violates those rules for simplicity:

```
int gob = 9;

constexpr void bad(int & arg)      // error: no return value
{
    ++arg;                         // error: modifies caller through argument
    glob = 7;                      // error: modifies nonlocal variable
}
```

If a compiler cannot determine that a **constexpr** function is “simple enough” (according to detailed rules in the standard), the function is considered an error.

8.6 Order of evaluation

The evaluation of a program – also called the execution of a program – proceeds through the statements according to the language rules. When this “thread of execution” reaches the definition of a variable, the variable is constructed; that is, memory is set aside for the object and the object is initialized. When the variable goes out of scope, the variable is destroyed; that is, the object it refers to is in principle removed and the compiler can use its memory for something else. For example:

```
string program_name = "silly";
vector<string> v;                           // v is global

void f()
{
    string s;                                // s is local to f
    while (cin>>s && s!="quit") {
        string stripped;                     // stripped is local to the loop
        string not_letters;
```

```

for (int i=0; i<s.size(); ++i)      // i has statement scope
    if (isalpha(s[i]))
        stripped += s[i];
    else
        not_letters += s[i];
    v.push_back(stripped);
    // ...
}
// ...
}

```

Global variables, such as **program_name** and **v**, are initialized before the first statement of **main()** is executed. They “live” until the program terminates, and then they are destroyed. They are constructed in the order in which they are defined (that is, **program_name** before **v**) and destroyed in the reverse order (that is, **v** before **program_name**).

When someone calls **f()**, first **s** is constructed; that is, **s** is initialized to the empty string. It will live until we return from **f()**.

Each time we enter the block that is the body of the **while**-statement, **stripped** and **not_letters** are constructed. Since **stripped** is defined before **not_letters**, **stripped** is constructed before **not_letters**. They live until the end of the loop, where they are destroyed in the reverse order of construction (that is, **not_letters** before **stripped**) before the condition is reevaluated. So, if ten strings are seen before we encounter the string **quit**, **stripped** and **not_letters** will each be constructed and destroyed ten times.

Each time we reach the **for**-statement, **i** is constructed. Each time we exit the **for**-statement, **i** is destroyed before we reach the **v.push_back(stripped);** statement.

Please note that compilers (and linkers) are clever beasts and they are allowed to – and do – optimize code as long as the results are equivalent to what we have described here. In particular, compilers are clever at not allocating and deallocating memory more often than is really necessary.

8.6.1 Expression evaluation



The order of evaluation of sub-expressions is governed by rules designed to please an optimizer rather than to make life simple for the programmer. That’s unfortunate, but you should avoid complicated expressions anyway, and there is a simple rule that can keep you out of trouble: if you change the value of a variable in an expression, don’t read or write it twice in that same expression. For example:

```

v[i] = ++i;          // don't: undefined order of evaluation
v[++i] = i;          // don't: undefined order of evaluation

```

```
int x = ++i + ++i;           // don't: undefined order of evaluation
cout << ++i << ' ' << i << '\n'; // don't: undefined order of evaluation
f(++i,++i);                 // don't: undefined order of evaluation
```

Unfortunately, not all compilers warn if you write such bad code; it's bad because you can't rely on the results being the same if you move your code to another computer, use a different compiler, or use a different optimizer setting. Compilers really differ for such code; just don't do it.

Note in particular that **=** (assignment) is considered just another operator in an expression, so there is no guarantee that the left-hand side of an assignment is evaluated before the right-hand side. That's why **v[++i] = i** is undefined.

8.6.2 Global initialization

Global variables (and namespace variables; see §8.7) in a single translation unit are initialized in the order in which they appear. For example:

```
// file f1.cpp
int x1 = 1;
int y1 = x1+2;      // y1 becomes 3
```

This initialization logically takes place “before the code in **main()** is executed.”

Using a global variable in anything but the most limited circumstances is usually not a good idea. We have mentioned the problem of the programmer having no really effective way of knowing which parts of a large program read and/or write a global variable (§8.4). Another problem is that the order of initialization of global variables in different translation units is not defined. For example:

```
// file f2.cpp
extern int y1;
int y2 = y1+2;      // y2 becomes 2 or 5
```

Such code is to be avoided for several reasons: it uses global variables, it gives the global variables short names, and it uses complicated initialization of the global variables. If the globals in file **f1.cpp** are initialized before the globals in **f2.cpp**, **y2** will be initialized to **5** (as a programmer might naively and reasonably expect). However, if the globals in file **f2.cpp** are initialized before the globals in **f1.cpp**, **y2** will be initialized to **2** (because the memory used for global variables is initialized to **0** before complicated initialization is attempted). Avoid such code, and be very suspicious when you see global variables with nontrivial initializers; consider any initializer that isn't a constant expression complicated.



But what do you do if you really need a global variable (or constant) with a complicated initializer? A plausible example would be that we wanted a default value for a **Date** type we were providing for a library supporting business transactions:

```
const Date default_date(1970,1,1);      // the default date is January 1, 1970
```

How would we know that **default_date** was never used before it was initialized? Basically, we can't know, so we shouldn't write that definition. The technique that we use most often is to call a function that returns the value. For example:

```
const Date default_date()           // return the default date
{
    return Date(1970,1,1);
}
```

This constructs the **Date** every time we call **default_date()**. That is often fine, but if **default_date()** is called often and it is expensive to construct **Date**, we'd like to construct the **Date** once only. That is done like this:

```
const Date& default_date()
{
    static const Date dd(1970,1,1);    // initialize dd first time we get here
    return dd;
}
```

The **static** local variable is initialized (constructed) only the first time its function is called. Note that we returned a reference to eliminate unnecessary copying and, in particular, we returned a **const** reference to prevent the calling function from accidentally changing the value. The arguments about how to pass an argument (§8.5.6) also apply to returning values.

8.7 Namespaces

We use blocks to organize code within a function (§8.4). We use classes to organize functions, data, and types into a type (Chapter 9). A function and a class both do two things for us:

- They allow us to define a number of “entities” without worrying that their names clash with other names in our program.
- They give us a name to refer to what we have defined.

What we lack so far is something to organize classes, functions, data, and types into an identifiable and named part of a program without defining a type. The language mechanism for such grouping of declarations is a *namespace*. For example, we might like to provide a graphics library with classes called **Color**, **Shape**, **Line**, **Function**, and **Text** (see Chapter 13):

```
namespace Graph_lib {
    struct Color {/* ... */};
    struct Shape {/* ... */};
    struct Line : Shape {/* ... */};
    struct Function : Shape {/* ... */};
    struct Text : Shape {/* ... */};
    // ...
    int gui_main() {/* ... */}
}
```

Most likely somebody else in the world has used those names, but now that doesn't matter. You might define something called **Text**, but our **Text** doesn't interfere. **Graph_lib::Text** is one of our classes and your **Text** is not. We have a problem only if you have a class or a namespace called **Graph_lib** with **Text** as its member. **Graph_lib** is a slightly ugly name; we chose it because the "pretty and obvious" name **Graphics** had a greater chance of already being used somewhere.

Let's say that your **Text** was part of a text manipulation library. The same logic that made us put our graphics facilities into namespace **Graph_lib** should make you put your text manipulation facilities into a namespace called something like **TextLib**:

```
namespace TextLib {
    class Text {/* ... */};
    class Glyph {/* ... */};
    class Line {/* ... */};
    // ...
}
```

If we both used the global namespace, we could have been in real trouble. Someone trying to use both of our libraries would have had really bad name clashes for **Text** and **Line**. Worse, if we both had users for our libraries we would not have been able to change our names, such as **Line** and **Text**, to avoid clashes. We avoided that problem by using namespaces; that is, our **Text** is **Graph_lib::Text** and yours is **TextLib::Text**. A name composed of a namespace name (or a class name) and a member name combined by **::** is called a *fully qualified name*.

8.7.1 using declarations and using directives

Writing fully qualified names can be tedious. For example, the facilities of the C++ standard library are defined in namespace **std** and can be used like this:

```
#include<string>      // get the string library
#include<iostream>     // get the iostream library

int main()
{
    std::string name;
    std::cout << "Please enter your first name\n";
    std::cin >> name;
    std::cout << "Hello, " << name << '\n';
}
```

Having seen the standard library **string** and **cout** thousands of times, we don't really want to have to refer to them by their "proper" fully qualified names **std::string** and **std::cout** all the time. A solution is to say that "by **string**, I mean **std::string**," "by **cout**, I mean **std::cout**," etc.:

```
using std::string;      // string means std::string
using std::cout;        // cout means std::cout
// . . .
```

That construct is called a **using** declaration; it is the programming equivalent to using plain "Greg" to refer to Greg Hansen, when there are no other Gregs in the room.

Sometimes, we prefer an even stronger "shorthand" for the use of names from a namespace: "If you don't find a declaration for a name in this scope, look in **std**." The way to say that is to use a **using** directive:

```
using namespace std;    // make names from std directly accessible
```

So we get this common style:

```
#include<string>      // get the string library
#include<iostream>     // get the iostream library
using namespace std;    // make names from std directly accessible

int main()
{
    string name;
    cout << "Please enter your first name\n";
```

```
    cin >> name;
    cout << "Hello, " << name << '\n';
}
```

The `cin` is `std::cin`, the `string` is `std::string`, etc. As long as you use `std_lib_facilities.h`, you don't need to worry about standard headers and the `std` namespace.

It is usually a good idea to avoid `using` directives for any namespace except for a namespace, such as `std`, that's extremely well known in an application area. The problem with overuse of `using` directives is that you lose track of which names come from where, so that you again start to get name clashes. Explicit qualification with namespace names and `using` declarations doesn't suffer from that problem. So, putting a `using` directive in a header file (so that users can't avoid it) is a very bad habit. However, to simplify our initial code we did place a `using` directive for `std` in `std_lib_facilities.h`. That allowed us to write

```
#include "std_lib_facilities.h"

int main()
{
    string name;
    cout << "Please enter your first name\n";
    cin >> name;
    cout << "Hello, " << name << '\n';
}
```

We promise never to do that for any namespace except `std`.



Drill

1. Create three files: `my.h`, `my.cpp`, and `use.cpp`. The header file `my.h` contains

```
extern int foo;
void print_foo();
void print(int);
```

The source code file `my.cpp` `#includes` `my.h` and `std_lib_facilities.h`, defines `print_foo()` to print the value of `foo` using `cout`, and `print(int i)` to print the value of `i` using `cout`.

The source code file `use.cpp` `#includes` `my.h`, defines `main()` to set the value of `foo` to `7` and print it using `print_foo()`, and to print the

value of **99** using **print()**. Note that **use.cpp** does not **#include std_lib_facilities.h** as it doesn't directly use any of those facilities.

Get these files compiled and run. On Windows, you need to have both **use.cpp** and **my.cpp** in a project and use **{ char cc; cin>>cc; }** in **use.cpp** to be able to see your output. Hint: You need to **#include <iostream>** to use **cin**.

2. Write three functions **swap_v(int,int)**, **swap_r(int&,int&)**, and **swap_cr(const int&, const int&)**. Each should have the body

```
{ int temp; temp = a, a=b; b=temp; }
```

where **a** and **b** are the names of the arguments.

Try calling each swap like this

```
int x = 7;
int y =9;
swap_?(x,y);           // replace ? by v, r, or cr
swap_?(7,9);
const int cx = 7;
const int cy = 9;
swap_?(cx,cy);
swap_?(7.7,9.9);
double dx = 7.7;
double dy = 9.9;
swap_?(dx,dy);
swap_?(7.7,9.9);
```

Which functions and calls compiled, and why? After each swap that compiled, print the value of the arguments after the call to see if they were actually swapped. If you are surprised by a result, consult §8.6.

3. Write a program using a single file containing three namespaces **X**, **Y**, and **Z** so that the following **main()** works correctly:

```
int main()
{
    X::var = 7;
    X::print();           // print X's var
    using namespace Y;
    var = 9;
    print();              // print Y's var
    {
        using Z::var;
        using Z::print;
        var = 11;
        print();          // print Z's var
    }
}
```

```
    print();           // print Y's var
    X::print();       // print X's var
}
```

Each namespace needs to define a variable called **var** and a function called **print()** that outputs the appropriate **var** using **cout**.

Review

1. What is the difference between a declaration and a definition?
2. How do we syntactically distinguish between a function declaration and a function definition?
3. How do we syntactically distinguish between a variable declaration and a variable definition?
4. Why can't you use the functions in the calculator program from Chapter 6 without declaring them first?
5. Is **int a;** a definition or just a declaration?
6. Why is it a good idea to initialize variables as they are declared?
7. What can a function declaration consist of?
8. What good does indentation do?
9. What are header files used for?
10. What is the scope of a declaration?
11. What kinds of scope are there? Give an example of each.
12. What is the difference between a class scope and local scope?
13. Why should a programmer minimize the number of global variables?
14. What is the difference between pass-by-value and pass-by-reference?
15. What is the difference between pass-by-reference and pass-by-**const**-reference?
16. What is a **swap()**?
17. Would you ever define a function with a **vector<double>**-by-value parameter?
18. Give an example of undefined order of evaluation. Why can undefined order of evaluation be a problem?
19. What do **x&&y** and **x||y**, respectively, mean?
20. Which of the following is standard-conforming C++: functions within functions, functions within classes, classes within classes, classes within functions?
21. What goes into an activation record?
22. What is a call stack and why do we need one?
23. What is the purpose of a namespace?
24. How does a namespace differ from a class?
25. What is a **using** declaration?

26. Why should you avoid **using** directives in a header?
27. What is namespace **std**?

Terms

activation record	function	pass-by-reference
argument	function definition	pass-by-value
argument passing	global scope	recursion
call stack	header file	return
class scope	initializer	return value
const	local scope	scope
constexpr	namespace	statement scope
declaration	namespace scope	technicalities
definition	nested block	undeclared identifier
extern	parameter	using declaration
forward declaration	pass-by- const -reference	using directive

Exercises

1. Modify the calculator program from Chapter 7 to make the input stream an explicit parameter (as shown in §8.5.8), rather than simply using **cin**. Also give the **Token_stream** constructor (§7.8.2) an **istream&** parameter so that when we figure out how to make our own **istreams** (e.g., attached to files), we can use the calculator for those. Hint: Don't try to copy an **istream**.
2. Write a function **print()** that prints a **vector** of **ints** to **cout**. Give it two arguments: a **string** for "labeling" the output and a **vector**.
3. Create a **vector** of Fibonacci numbers and print them using the function from exercise 2. To create the **vector**, write a function, **fibonacci(x,y,v,n)**, where integers **x** and **y** are **ints**, **v** is an empty **vector<int>**, and **n** is the number of elements to put into **v**; **v[0]** will be **x** and **v[1]** will be **y**. A Fibonacci number is one that is part of a sequence where each element is the sum of the two previous ones. For example, starting with 1 and 2, we get 1, 2, 3, 5, 8, 13, 21, . . . Your **fibonacci()** function should make such a sequence starting with its **x** and **y** arguments.
4. An **int** can hold integers only up to a maximum number. Find an approximation of that maximum number by using **fibonacci()**.
5. Write two functions that reverse the order of elements in a **vector<int>**. For example, 1, 3, 5, 7, 9 becomes 9, 7, 5, 3, 1. The first reverse function should produce a new **vector** with the reversed sequence, leaving its original **vector** unchanged. The other reverse function should reverse the elements of its **vector** without using any other **vectors** (hint: **swap**).

6. Write versions of the functions from exercise 5, but with a `vector<string>`.
7. Read five names into a `vector<string> name`, then prompt the user for the ages of the people named and store the ages in a `vector<double> age`. Then print out the five (`name[i],age[i]`) pairs. Sort the names (`sort(name.begin(),name.end())`) and print out the (`name[i],age[i]`) pairs. The tricky part here is to get the `age vector` in the correct order to match the sorted `name vector`. Hint: Before sorting `name`, take a copy and use that to make a copy of `age` in the right order after sorting `name`. Then, do that exercise again but allowing an arbitrary number of names.
9. Write a function that given two `vector<double>`s `price` and `weight` computes a value (an “index”) that is the sum of all `price[i]*weight[i]`. Make sure to have `weight.size()==price.size()`.
10. Write a function `maxv()` that returns the largest element of a `vector` argument.
11. Write a function that finds the smallest and the largest element of a `vector` argument and also computes the mean and the median. Do not use global variables. Either return a `struct` containing the results or pass them back through reference arguments. Which of the two ways of returning several result values do you prefer and why?
12. Improve `print_until_s()` from §8.5.2. Test it. What makes a good set of test cases? Give reasons. Then, write a `print_until_ss()` that prints until it sees a second occurrence of its `quit` argument.
13. Write a function that takes a `vector<string>` argument and returns a `vector<int>` containing the number of characters in each `string`. Also find the longest and the shortest `string` and the lexicographically first and last `string`. How many separate functions would you use for these tasks? Why?
14. Can we declare a non-reference function argument `const` (e.g., `void f(const int);`)? What might that mean? Why might we want to do that? Why don’t people do that often? Try it; write a couple of small programs to see what works.

Postscript

We could have put much of this chapter (and much of the next) into an appendix. However, you’ll need most of the facilities described here in Part II of this book. You’ll also encounter most of the problems that these facilities were invented to help solve very soon. Most simple programming projects that you might undertake will require you to solve such problems. So, to save time and minimize confusion, a somewhat systematic approach is called for, rather than a series of “random” visits to manuals and appendices.

