

Bestseller Since 1986

Completely Rewritten for the New C++11 Standard



Fifth Edition

# C++ Primer

Stanley B. Lippman  
Josée Lajoie  
Barbara Moo

*C++ Primer*  
*Fifth Edition*

*This page intentionally left blank*

# *C++ Primer* *Fifth Edition*

Stanley B. Lippman  
Josée Lajoie  
Barbara E. Moo

◆◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco  
New York • Toronto • Montreal • London • Munich • Paris • Madrid  
Capetown • Sidney • 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.

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

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

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

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

International Sales  
international@pearsoned.com

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

*Library of Congress Cataloging-in-Publication Data*

Lippman, Stanley B.

C++ primer / Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. – 5th ed.

p. cm.

Includes index.

ISBN 0-321-71411-3 (pbk. : alk. paper) 1. C++ (Computer program language) I. Lajoie, Josée. II. Moo, Barbara E. III. Title.

QA76.73.C153L57697 2013

005.13'3—dc23

2012020184

Copyright © 2013 Objectwrite Inc., Josée Lajoie and Barbara E. Moo

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.

C++ Primer, Fifth Edition, features an enhanced, layflat binding, which allows the book to stay open more easily when placed on a flat surface. This special binding method—notable by a small space inside the spine—also increases durability.

ISBN-13: 978-0-321-71411-4

ISBN-10: 0-321-71411-3

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

Sixth printing, May 2015

*To Beth,  
who makes this,  
and all things,  
possible.*

---

*To Daniel and Anna,  
who contain  
virtually  
all possibilities.*  
—SBL

*To Mark and Mom,  
for their  
unconditional love and support.*  
—JL

*To Andy,  
who taught me  
to program  
and so much more.*  
—BEM

*This page intentionally left blank*

# Contents

<b>Preface</b>	<b>xxiii</b>
<b>Chapter 1 Getting Started</b>	<b>1</b>
1.1 Writing a Simple C++ Program	2
1.1.1 Compiling and Executing Our Program	3
1.2 A First Look at Input/Output	5
1.3 A Word about Comments	9
1.4 Flow of Control	11
1.4.1 The while Statement	11
1.4.2 The for Statement	13
1.4.3 Reading an Unknown Number of Inputs	14
1.4.4 The if Statement	17
1.5 Introducing Classes	19
1.5.1 The Sales_item Class	20
1.5.2 A First Look at Member Functions	23
1.6 The Bookstore Program	24
Chapter Summary	26
Defined Terms	26
 <b>Part I The Basics</b>	 <b>29</b>
<b>Chapter 2 Variables and Basic Types</b>	<b>31</b>
2.1 Primitive Built-in Types	32
2.1.1 Arithmetic Types	32
2.1.2 Type Conversions	35
2.1.3 Literals	38
2.2 Variables	41
2.2.1 Variable Definitions	41
2.2.2 Variable Declarations and Definitions	44
2.2.3 Identifiers	46
2.2.4 Scope of a Name	48
2.3 Compound Types	50
2.3.1 References	50
2.3.2 Pointers	52



2.3.3	Understanding Compound Type Declarations . . . . .	57
2.4	const Qualifier . . . . .	59
2.4.1	References to const . . . . .	61
2.4.2	Pointers and const . . . . .	62
2.4.3	Top-Level const . . . . .	63
2.4.4	constexpr and Constant Expressions . . . . .	65
2.5	Dealing with Types . . . . .	67
2.5.1	Type Aliases . . . . .	67
2.5.2	The auto Type Specifier . . . . .	68
2.5.3	The decltype Type Specifier . . . . .	70
2.6	Defining Our Own Data Structures . . . . .	72
2.6.1	Defining the Sales_data Type . . . . .	72
2.6.2	Using the Sales_data Class . . . . .	74
2.6.3	Writing Our Own Header Files . . . . .	76
	Chapter Summary . . . . .	78
	Defined Terms . . . . .	78
<b>Chapter 3 Strings, Vectors, and Arrays . . . . .</b>		<b>81</b>
3.1	Namespace using Declarations . . . . .	82
3.2	Library string Type . . . . .	84
3.2.1	Defining and Initializing strings . . . . .	84
3.2.2	Operations on strings . . . . .	85
3.2.3	Dealing with the Characters in a string . . . . .	90
3.3	Library vector Type . . . . .	96
3.3.1	Defining and Initializing vectors . . . . .	97
3.3.2	Adding Elements to a vector . . . . .	100
3.3.3	Other vector Operations . . . . .	102
3.4	Introducing Iterators . . . . .	106
3.4.1	Using Iterators . . . . .	106
3.4.2	Iterator Arithmetic . . . . .	111
3.5	Arrays . . . . .	113
3.5.1	Defining and Initializing Built-in Arrays . . . . .	113
3.5.2	Accessing the Elements of an Array . . . . .	116
3.5.3	Pointers and Arrays . . . . .	117
3.5.4	C-Style Character Strings . . . . .	122
3.5.5	Interfacing to Older Code . . . . .	124
3.6	Multidimensional Arrays . . . . .	125
	Chapter Summary . . . . .	131
	Defined Terms . . . . .	131
<b>Chapter 4 Expressions . . . . .</b>		<b>133</b>
4.1	Fundamentals . . . . .	134
4.1.1	Basic Concepts . . . . .	134
4.1.2	Precedence and Associativity . . . . .	136
4.1.3	Order of Evaluation . . . . .	137
4.2	Arithmetic Operators . . . . .	139
4.3	Logical and Relational Operators . . . . .	141

4.4	Assignment Operators . . . . .	144
4.5	Increment and Decrement Operators . . . . .	147
4.6	The Member Access Operators . . . . .	150
4.7	The Conditional Operator . . . . .	151
4.8	The Bitwise Operators . . . . .	152
4.9	The sizeof Operator . . . . .	156
4.10	Comma Operator . . . . .	157
4.11	Type Conversions . . . . .	159
4.11.1	The Arithmetic Conversions . . . . .	159
4.11.2	Other Implicit Conversions . . . . .	161
4.11.3	Explicit Conversions . . . . .	162
4.12	Operator Precedence Table . . . . .	166
	Chapter Summary . . . . .	168
	Defined Terms . . . . .	168
<b>Chapter 5</b>	<b>Statements . . . . .</b>	<b>171</b>
5.1	Simple Statements . . . . .	172
5.2	Statement Scope . . . . .	174
5.3	Conditional Statements . . . . .	174
5.3.1	The if Statement . . . . .	175
5.3.2	The switch Statement . . . . .	178
5.4	Iterative Statements . . . . .	183
5.4.1	The while Statement . . . . .	183
5.4.2	Traditional for Statement . . . . .	185
5.4.3	Range for Statement . . . . .	187
5.4.4	The do while Statement . . . . .	189
5.5	Jump Statements . . . . .	190
5.5.1	The break Statement . . . . .	190
5.5.2	The continue Statement . . . . .	191
5.5.3	The goto Statement . . . . .	192
5.6	try Blocks and Exception Handling . . . . .	193
5.6.1	A throw Expression . . . . .	193
5.6.2	The try Block . . . . .	194
5.6.3	Standard Exceptions . . . . .	197
	Chapter Summary . . . . .	199
	Defined Terms . . . . .	199
<b>Chapter 6</b>	<b>Functions . . . . .</b>	<b>201</b>
6.1	Function Basics . . . . .	202
6.1.1	Local Objects . . . . .	204
6.1.2	Function Declarations . . . . .	206
6.1.3	Separate Compilation . . . . .	207
6.2	Argument Passing . . . . .	208
6.2.1	Passing Arguments by Value . . . . .	209
6.2.2	Passing Arguments by Reference . . . . .	210
6.2.3	const Parameters and Arguments . . . . .	212
6.2.4	Array Parameters . . . . .	214

6.2.5	main: Handling Command-Line Options . . . . .	218
6.2.6	Functions with Varying Parameters . . . . .	220
6.3	Return Types and the <code>return</code> Statement . . . . .	222
6.3.1	Functions with No Return Value . . . . .	223
6.3.2	Functions That Return a Value . . . . .	223
6.3.3	Returning a Pointer to an Array . . . . .	228
6.4	Overloaded Functions . . . . .	230
6.4.1	Overloading and Scope . . . . .	234
6.5	Features for Specialized Uses . . . . .	236
6.5.1	Default Arguments . . . . .	236
6.5.2	Inline and <code>constexpr</code> Functions . . . . .	238
6.5.3	Aids for Debugging . . . . .	240
6.6	Function Matching . . . . .	242
6.6.1	Argument Type Conversions . . . . .	245
6.7	Pointers to Functions . . . . .	247
	Chapter Summary . . . . .	251
	Defined Terms . . . . .	251
<b>Chapter 7</b>	<b>Classes . . . . .</b>	<b>253</b>
7.1	Defining Abstract Data Types . . . . .	254
7.1.1	Designing the <code>Sales_data</code> Class . . . . .	254
7.1.2	Defining the Revised <code>Sales_data</code> Class . . . . .	256
7.1.3	Defining Nonmember Class-Related Functions . . . . .	260
7.1.4	Constructors . . . . .	262
7.1.5	Copy, Assignment, and Destruction . . . . .	267
7.2	Access Control and Encapsulation . . . . .	268
7.2.1	Friends . . . . .	269
7.3	Additional Class Features . . . . .	271
7.3.1	Class Members Revisited . . . . .	271
7.3.2	Functions That Return <code>*this</code> . . . . .	275
7.3.3	Class Types . . . . .	277
7.3.4	Friendship Revisited . . . . .	279
7.4	Class Scope . . . . .	282
7.4.1	Name Lookup and Class Scope . . . . .	283
7.5	Constructors Revisited . . . . .	288
7.5.1	Constructor Initializer List . . . . .	288
7.5.2	Delegating Constructors . . . . .	291
7.5.3	The Role of the Default Constructor . . . . .	293
7.5.4	Implicit Class-Type Conversions . . . . .	294
7.5.5	Aggregate Classes . . . . .	298
7.5.6	Literal Classes . . . . .	299
7.6	<code>static</code> Class Members . . . . .	300
	Chapter Summary . . . . .	305
	Defined Terms . . . . .	305

**Part II The C++ Library 307**

**Chapter 8 The IO Library . . . . . 309**

8.1 The IO Classes . . . . . 310

8.1.1 No Copy or Assign for IO Objects . . . . . 311

8.1.2 Condition States . . . . . 312

8.1.3 Managing the Output Buffer . . . . . 314

8.2 File Input and Output . . . . . 316

8.2.1 Using File Stream Objects . . . . . 317

8.2.2 File Modes . . . . . 319

8.3 string Streams . . . . . 321

8.3.1 Using an `istringstream` . . . . . 321

8.3.2 Using `ostringstreams` . . . . . 323

Chapter Summary . . . . . 324

Defined Terms . . . . . 324

**Chapter 9 Sequential Containers . . . . . 325**

9.1 Overview of the Sequential Containers . . . . . 326

9.2 Container Library Overview . . . . . 328

9.2.1 Iterators . . . . . 331

9.2.2 Container Type Members . . . . . 332

9.2.3 `begin` and `end` Members . . . . . 333

9.2.4 Defining and Initializing a Container . . . . . 334

9.2.5 Assignment and `swap` . . . . . 337

9.2.6 Container Size Operations . . . . . 340

9.2.7 Relational Operators . . . . . 340

9.3 Sequential Container Operations . . . . . 341

9.3.1 Adding Elements to a Sequential Container . . . . . 341

9.3.2 Accessing Elements . . . . . 346

9.3.3 Erasing Elements . . . . . 348

9.3.4 Specialized `forward_list` Operations . . . . . 350

9.3.5 Resizing a Container . . . . . 352

9.3.6 Container Operations May Invalidate Iterators . . . . . 353

9.4 How a `vector` Grows . . . . . 355

9.5 Additional `string` Operations . . . . . 360

9.5.1 Other Ways to Construct `strings` . . . . . 360

9.5.2 Other Ways to Change a `string` . . . . . 361

9.5.3 `string` Search Operations . . . . . 364

9.5.4 The `compare` Functions . . . . . 366

9.5.5 Numeric Conversions . . . . . 367

9.6 Container Adaptors . . . . . 368

Chapter Summary . . . . . 372

Defined Terms . . . . . 372

<b>Chapter 10 Generic Algorithms</b>	<b>375</b>
10.1 Overview	376
10.2 A First Look at the Algorithms	378
10.2.1 Read-Only Algorithms	379
10.2.2 Algorithms That Write Container Elements	380
10.2.3 Algorithms That Reorder Container Elements	383
10.3 Customizing Operations	385
10.3.1 Passing a Function to an Algorithm	386
10.3.2 Lambda Expressions	387
10.3.3 Lambda Captures and Returns	392
10.3.4 Binding Arguments	397
10.4 Revisiting Iterators	401
10.4.1 Insert Iterators	401
10.4.2 <code>iostream</code> Iterators	403
10.4.3 Reverse Iterators	407
10.5 Structure of Generic Algorithms	410
10.5.1 The Five Iterator Categories	410
10.5.2 Algorithm Parameter Patterns	412
10.5.3 Algorithm Naming Conventions	413
10.6 Container-Specific Algorithms	415
Chapter Summary	417
Defined Terms	417
<b>Chapter 11 Associative Containers</b>	<b>419</b>
11.1 Using an Associative Container	420
11.2 Overview of the Associative Containers	423
11.2.1 Defining an Associative Container	423
11.2.2 Requirements on Key Type	424
11.2.3 The <code>pair</code> Type	426
11.3 Operations on Associative Containers	428
11.3.1 Associative Container Iterators	429
11.3.2 Adding Elements	431
11.3.3 Erasing Elements	434
11.3.4 Subscripting a <code>map</code>	435
11.3.5 Accessing Elements	436
11.3.6 A Word Transformation Map	440
11.4 The Unordered Containers	443
Chapter Summary	447
Defined Terms	447
<b>Chapter 12 Dynamic Memory</b>	<b>449</b>
12.1 Dynamic Memory and Smart Pointers	450
12.1.1 The <code>shared_ptr</code> Class	450
12.1.2 Managing Memory Directly	458
12.1.3 Using <code>shared_ptrs</code> with <code>new</code>	464
12.1.4 Smart Pointers and Exceptions	467
12.1.5 <code>unique_ptr</code>	470

12.1.6	<code>weak_ptr</code> . . . . .	473
12.2	Dynamic Arrays . . . . .	476
12.2.1	<code>new</code> and Arrays . . . . .	477
12.2.2	The <code>allocator</code> Class . . . . .	481
12.3	Using the Library: A Text-Query Program . . . . .	484
12.3.1	Design of the Query Program . . . . .	485
12.3.2	Defining the Query Program Classes . . . . .	487
	Chapter Summary . . . . .	491
	Defined Terms . . . . .	491

## Part III Tools for Class Authors 493

### Chapter 13 Copy Control . . . . . 495

13.1	Copy, Assign, and Destroy . . . . .	496
13.1.1	The Copy Constructor . . . . .	496
13.1.2	The Copy-Assignment Operator . . . . .	500
13.1.3	The Destructor . . . . .	501
13.1.4	The Rule of Three/Five . . . . .	503
13.1.5	Using <code>= default</code> . . . . .	506
13.1.6	Preventing Copies . . . . .	507
13.2	Copy Control and Resource Management . . . . .	510
13.2.1	Classes That Act Like Values . . . . .	511
13.2.2	Defining Classes That Act Like Pointers . . . . .	513
13.3	Swap . . . . .	516
13.4	A Copy-Control Example . . . . .	519
13.5	Classes That Manage Dynamic Memory . . . . .	524
13.6	Moving Objects . . . . .	531
13.6.1	Rvalue References . . . . .	532
13.6.2	Move Constructor and Move Assignment . . . . .	534
13.6.3	Rvalue References and Member Functions . . . . .	544
	Chapter Summary . . . . .	549
	Defined Terms . . . . .	549

### Chapter 14 Overloaded Operations and Conversions . . . . . 551

14.1	Basic Concepts . . . . .	552
14.2	Input and Output Operators . . . . .	556
14.2.1	Overloading the Output Operator <code>&lt;&lt;</code> . . . . .	557
14.2.2	Overloading the Input Operator <code>&gt;&gt;</code> . . . . .	558
14.3	Arithmetic and Relational Operators . . . . .	560
14.3.1	Equality Operators . . . . .	561
14.3.2	Relational Operators . . . . .	562
14.4	Assignment Operators . . . . .	563
14.5	Subscript Operator . . . . .	564
14.6	Increment and Decrement Operators . . . . .	566
14.7	Member Access Operators . . . . .	569
14.8	Function-Call Operator . . . . .	571

- 14.8.1 Lambdas Are Function Objects . . . . . 572
  - 14.8.2 Library-Defined Function Objects . . . . . 574
  - 14.8.3 Callable Objects and function . . . . . 576
- 14.9 Overloading, Conversions, and Operators . . . . . 579
  - 14.9.1 Conversion Operators . . . . . 580
  - 14.9.2 Avoiding Ambiguous Conversions . . . . . 583
  - 14.9.3 Function Matching and Overloaded Operators . . . . . 587
- Chapter Summary . . . . . 590
- Defined Terms . . . . . 590
- Chapter 15 Object-Oriented Programming . . . . . 591**
  - 15.1 OOP: An Overview . . . . . 592
  - 15.2 Defining Base and Derived Classes . . . . . 594
    - 15.2.1 Defining a Base Class . . . . . 594
    - 15.2.2 Defining a Derived Class . . . . . 596
    - 15.2.3 Conversions and Inheritance . . . . . 601
  - 15.3 Virtual Functions . . . . . 603
  - 15.4 Abstract Base Classes . . . . . 608
  - 15.5 Access Control and Inheritance . . . . . 611
  - 15.6 Class Scope under Inheritance . . . . . 617
  - 15.7 Constructors and Copy Control . . . . . 622
    - 15.7.1 Virtual Destructors . . . . . 622
    - 15.7.2 Synthesized Copy Control and Inheritance . . . . . 623
    - 15.7.3 Derived-Class Copy-Control Members . . . . . 625
    - 15.7.4 Inherited Constructors . . . . . 628
  - 15.8 Containers and Inheritance . . . . . 630
    - 15.8.1 Writing a Basket Class . . . . . 631
  - 15.9 Text Queries Revisited . . . . . 634
    - 15.9.1 An Object-Oriented Solution . . . . . 636
    - 15.9.2 The query\_base and Query Classes . . . . . 639
    - 15.9.3 The Derived Classes . . . . . 642
    - 15.9.4 The eval Functions . . . . . 645
  - Chapter Summary . . . . . 649
  - Defined Terms . . . . . 649
- Chapter 16 Templates and Generic Programming . . . . . 651**
  - 16.1 Defining a Template . . . . . 652
    - 16.1.1 Function Templates . . . . . 652
    - 16.1.2 Class Templates . . . . . 658
    - 16.1.3 Template Parameters . . . . . 668
    - 16.1.4 Member Templates . . . . . 672
    - 16.1.5 Controlling Instantiations . . . . . 675
    - 16.1.6 Efficiency and Flexibility . . . . . 676
  - 16.2 Template Argument Deduction . . . . . 678
    - 16.2.1 Conversions and Template Type Parameters . . . . . 679
    - 16.2.2 Function-Template Explicit Arguments . . . . . 681
    - 16.2.3 Trailing Return Types and Type Transformation . . . . . 683

- 16.2.4 Function Pointers and Argument Deduction . . . . . 686
- 16.2.5 Template Argument Deduction and References . . . . . 687
- 16.2.6 Understanding `std::move` . . . . . 690
- 16.2.7 Forwarding . . . . . 692
- 16.3 Overloading and Templates . . . . . 694
- 16.4 Variadic Templates . . . . . 699
  - 16.4.1 Writing a Variadic Function Template . . . . . 701
  - 16.4.2 Pack Expansion . . . . . 702
  - 16.4.3 Forwarding Parameter Packs . . . . . 704
- 16.5 Template Specializations . . . . . 706
- Chapter Summary . . . . . 713
- Defined Terms . . . . . 713

**Part IV    Advanced Topics** **715**

**Chapter 17 Specialized Library Facilities** . . . . . **717**

- 17.1 The tuple Type . . . . . 718
  - 17.1.1 Defining and Initializing tuples . . . . . 718
  - 17.1.2 Using a tuple to Return Multiple Values . . . . . 721
- 17.2 The bitset Type . . . . . 723
  - 17.2.1 Defining and Initializing bitsets . . . . . 723
  - 17.2.2 Operations on bitsets . . . . . 725
- 17.3 Regular Expressions . . . . . 728
  - 17.3.1 Using the Regular Expression Library . . . . . 729
  - 17.3.2 The Match and Regex Iterator Types . . . . . 734
  - 17.3.3 Using Subexpressions . . . . . 738
  - 17.3.4 Using `regex_replace` . . . . . 741
- 17.4 Random Numbers . . . . . 745
  - 17.4.1 Random-Number Engines and Distribution . . . . . 745
  - 17.4.2 Other Kinds of Distributions . . . . . 749
- 17.5 The IO Library Revisited . . . . . 752
  - 17.5.1 Formatted Input and Output . . . . . 753
  - 17.5.2 Unformatted Input/Output Operations . . . . . 761
  - 17.5.3 Random Access to a Stream . . . . . 763
- Chapter Summary . . . . . 769
- Defined Terms . . . . . 769

**Chapter 18 Tools for Large Programs** . . . . . **771**

- 18.1 Exception Handling . . . . . 772
  - 18.1.1 Throwing an Exception . . . . . 772
  - 18.1.2 Catching an Exception . . . . . 775
  - 18.1.3 Function `try` Blocks and Constructors . . . . . 777
  - 18.1.4 The `noexcept` Exception Specification . . . . . 779
  - 18.1.5 Exception Class Hierarchies . . . . . 782
- 18.2 Namespaces . . . . . 785
  - 18.2.1 Namespace Definitions . . . . . 785



- 18.2.2 Using Namespace Members . . . . . 792
  - 18.2.3 Classes, Namespaces, and Scope . . . . . 796
  - 18.2.4 Overloading and Namespaces . . . . . 800
- 18.3 Multiple and Virtual Inheritance . . . . . 802
  - 18.3.1 Multiple Inheritance . . . . . 803
  - 18.3.2 Conversions and Multiple Base Classes . . . . . 805
  - 18.3.3 Class Scope under Multiple Inheritance . . . . . 807
  - 18.3.4 Virtual Inheritance . . . . . 810
  - 18.3.5 Constructors and Virtual Inheritance . . . . . 813
- Chapter Summary . . . . . 816
- Defined Terms . . . . . 816

**Chapter 19 Specialized Tools and Techniques . . . . . 819**

- 19.1 Controlling Memory Allocation . . . . . 820
  - 19.1.1 Overloading new and delete . . . . . 820
  - 19.1.2 Placement new Expressions . . . . . 823
- 19.2 Run-Time Type Identification . . . . . 825
  - 19.2.1 The dynamic\_cast Operator . . . . . 825
  - 19.2.2 The typeid Operator . . . . . 826
  - 19.2.3 Using RTTI . . . . . 828
  - 19.2.4 The type\_info Class . . . . . 831
- 19.3 Enumerations . . . . . 832
- 19.4 Pointer to Class Member . . . . . 835
  - 19.4.1 Pointers to Data Members . . . . . 836
  - 19.4.2 Pointers to Member Functions . . . . . 838
  - 19.4.3 Using Member Functions as Callable Objects . . . . . 841
- 19.5 Nested Classes . . . . . 843
- 19.6 union: A Space-Saving Class . . . . . 847
- 19.7 Local Classes . . . . . 852
- 19.8 Inherently Nonportable Features . . . . . 854
  - 19.8.1 Bit-fields . . . . . 854
  - 19.8.2 volatile Qualifier . . . . . 856
  - 19.8.3 Linkage Directives: extern "C" . . . . . 857
- Chapter Summary . . . . . 862
- Defined Terms . . . . . 862

**Appendix A The Library . . . . . 865**

- A.1 Library Names and Headers . . . . . 866
- A.2 A Brief Tour of the Algorithms . . . . . 870
  - A.2.1 Algorithms to Find an Object . . . . . 871
  - A.2.2 Other Read-Only Algorithms . . . . . 872
  - A.2.3 Binary Search Algorithms . . . . . 873
  - A.2.4 Algorithms That Write Container Elements . . . . . 873
  - A.2.5 Partitioning and Sorting Algorithms . . . . . 875
  - A.2.6 General Reordering Operations . . . . . 877
  - A.2.7 Permutation Algorithms . . . . . 879
  - A.2.8 Set Algorithms for Sorted Sequences . . . . . 880

---

A.2.9	Minimum and Maximum Values . . . . .	880
A.2.10	Numeric Algorithms . . . . .	881
A.3	Random Numbers . . . . .	882
A.3.1	Random Number Distributions . . . . .	883
A.3.2	Random Number Engines . . . . .	884
<b>Index</b>		<b>887</b>

*This page intentionally left blank*

# New Features in C++11

- 2.1.1 long long Type . . . . . 33
- 2.2.1 List Initialization . . . . . 43
- 2.3.2 nullptr Literal . . . . . 54
- 2.4.4 constexpr Variables . . . . . 66
- 2.5.1 Type Alias Declarations . . . . . 68
- 2.5.2 The auto Type Specifier . . . . . 68
- 2.5.3 The decltype Type Specifier . . . . . 70
- 2.6.1 In-Class Initializers . . . . . 73
- 3.2.2 Using auto or decltype for Type Abbreviation . . . . . 88
- 3.2.3 Range for Statement . . . . . 91
- 3.3 Defining a vector of vectors . . . . . 97
- 3.3.1 List Initialization for vectors . . . . . 98
- 3.4.1 Container cbegin and cend Functions . . . . . 109
- 3.5.3 Library begin and end Functions . . . . . 118
- 3.6 Using auto or decltype to Simplify Declarations . . . . . 129
- 4.2 Rounding Rules for Division . . . . . 141
- 4.4 Assignment from a Braced List of Values . . . . . 145
- 4.9 sizeof Applied to a Class Member . . . . . 157
- 5.4.3 Range for Statement . . . . . 187
- 6.2.6 Library initializer\_list Class . . . . . 220
- 6.3.2 List Initializing a Return Value . . . . . 226
- 6.3.3 Declaring a Trailing Return Type . . . . . 229
- 6.3.3 Using decltype to Simplify Return Type Declarations . . . . 230
- 6.5.2 constexpr Functions . . . . . 239
- 7.1.4 Using = default to Generate a Default Constructor . . . . . 265
- 7.3.1 In-class Initializers for Members of Class Type . . . . . 274
- 7.5.2 Delegating Constructors . . . . . 291
- 7.5.6 constexpr Constructors . . . . . 299
- 8.2.1 Using strings for File Names . . . . . 317
- 9.1 The array and forward\_list Containers . . . . . 327
- 9.2.3 Container cbegin and cend Functions . . . . . 334
- 9.2.4 List Initialization for Containers . . . . . 336
- 9.2.5 Container Nonmember swap Functions . . . . . 339
- 9.3.1 Return Type for Container insert Members . . . . . 344
- 9.3.1 Container emplace Members . . . . . 345

9.4	<code>shrink_to_fit</code> . . . . .	357
9.5.5	Numeric Conversion Functions for <code>strings</code> . . . . .	367
10.3.2	Lambda Expressions . . . . .	388
10.3.3	Trailing Return Type in Lambda Expressions . . . . .	396
10.3.4	The Library <code>bind</code> Function . . . . .	397
11.2.1	List Initialization of an Associative Container . . . . .	423
11.2.3	List Initializing <code>pair</code> Return Type . . . . .	427
11.3.2	List Initialization of a <code>pair</code> . . . . .	431
11.4	The Unordered Containers . . . . .	443
12.1	Smart Pointers . . . . .	450
12.1.1	The <code>shared_ptr</code> Class . . . . .	450
12.1.2	List Initialization of Dynamically Allocated Objects . . . . .	459
12.1.2	<code>auto</code> and Dynamic Allocation . . . . .	459
12.1.5	The <code>unique_ptr</code> Class . . . . .	470
12.1.6	The <code>weak_ptr</code> Class . . . . .	473
12.2.1	Range for Doesn't Apply to Dynamically Allocated Arrays . . . . .	477
12.2.1	List Initialization of Dynamically Allocated Arrays . . . . .	478
12.2.1	<code>auto</code> Can't Be Used to Allocate an Array . . . . .	478
12.2.2	<code>allocator::construct</code> Can Use any Constructor . . . . .	482
13.1.5	Using <code>= default</code> for Copy-Control Members . . . . .	506
13.1.6	Using <code>= delete</code> to Prevent Copying Class Objects . . . . .	507
13.5	Moving Instead of Copying Class Objects . . . . .	529
13.6.1	Rvalue References . . . . .	532
13.6.1	The Library <code>move</code> Function . . . . .	533
13.6.2	Move Constructor and Move Assignment . . . . .	534
13.6.2	Move Constructors Usually Should Be <code>noexcept</code> . . . . .	535
13.6.2	Move Iterators . . . . .	543
13.6.3	Reference Qualified Member Functions . . . . .	546
14.8.3	The function Class Template . . . . .	577
14.9.1	<code>explicit</code> Conversion Operators . . . . .	582
15.2.2	<code>override</code> Specifier for Virtual Functions . . . . .	596
15.2.2	Preventing Inheritance by Defining a Class as <code>final</code> . . . . .	600
15.3	<code>override</code> and <code>final</code> Specifiers for Virtual Functions . . . . .	606
15.7.2	Deleted Copy Control and Inheritance . . . . .	624
15.7.4	Inherited Constructors . . . . .	628
16.1.2	Declaring a Template Type Parameter as a Friend . . . . .	666
16.1.2	Template Type Aliases . . . . .	666
16.1.3	Default Template Arguments for Template Functions . . . . .	670
16.1.5	Explicit Control of Instantiation . . . . .	675
16.2.3	Template Functions and Trailing Return Types . . . . .	684
16.2.5	Reference Collapsing Rules . . . . .	688
16.2.6	<code>static_cast</code> from an Lvalue to an Rvalue . . . . .	691
16.2.7	The Library <code>forward</code> Function . . . . .	694
16.4	Variadic Templates . . . . .	699
16.4	The <code>sizeof... Operator</code> . . . . .	700
16.4.3	Variadic Templates and Forwarding . . . . .	704

- 17.1 The Library Tuple Class Template . . . . . 718
- 17.2.2 New bitset Operations . . . . . 726
- 17.3 The Regular Expression Library . . . . . 728
- 17.4 The Random Number Library . . . . . 745
- 17.5.1 Floating-Point Format Control . . . . . 757
- 18.1.4 The noexcept Exception Specifier . . . . . 779
- 18.1.4 The noexcept Operator . . . . . 780
- 18.2.1 Inline Namespaces . . . . . 790
- 18.3.1 Inherited Constructors and Multiple Inheritance . . . . . 804
- 19.3 Scoped enums . . . . . 832
- 19.3 Specifying the Type Used to Hold an enum . . . . . 834
- 19.3 Forward Declarations for enums . . . . . 834
- 19.4.3 The Library mem\_fn Class Template . . . . . 843
- 19.6 Union Members of Class Types . . . . . 848

*This page intentionally left blank*

# Preface

*Countless programmers* have learned C++ from previous editions of *C++ Primer*. During that time, C++ has matured greatly: Its focus, and that of its programming community, has widened from looking mostly at *machine* efficiency to devoting more attention to *programmer* efficiency.

In 2011, the C++ standards committee issued a major revision to the ISO C++ standard. This revised standard is latest step in C++'s evolution and continues the emphasis on programmer efficiency. The primary goals of the new standard are to

- Make the language more uniform and easier to teach and to learn
- Make the standard libraries easier, safer, and more efficient to use
- Make it easier to write efficient abstractions and libraries

In this edition, we have completely revised the *C++ Primer* to use the latest standard. You can get an idea of how extensively the new standard has affected C++ by reviewing the New Features Table of Contents, which lists the sections that cover new material and appears on page xxi.

Some additions in the new standard, such as `auto` for type inference, are pervasive. These facilities make the code in this edition easier to read and to understand. Programs (and programmers!) can ignore type details, which makes it easier to concentrate on what the program is intended to do. Other new features, such as smart pointers and move-enabled containers, let us write more sophisticated classes without having to contend with the intricacies of resource management. As a result, we can start to teach how to write your own classes much earlier in the book than we did in the Fourth Edition. We—and you—no longer have to worry about many of the details that stood in our way under the previous standard.

We've marked those parts of the text that cover features defined by the new standard, with a marginal icon. We hope that readers who are already familiar with the core of C++ will find these alerts useful in deciding where to focus their attention. We also expect that these icons will help explain error messages from compilers that might not yet support every new feature. Although nearly all of the examples in this book have been compiled under the current release of the GNU compiler, we realize some readers will not yet have access to completely updated compilers. Even though numerous capabilities have been added by the latest standard, the core language remains unchanged and forms the bulk of the material that we cover. Readers can use these icons to note which capabilities may not yet be available in their compiler.





## Why Read This Book?

Modern C++ can be thought of as comprising three parts:

- The low-level language, much of which is inherited from C
- More advanced language features that allow us to define our own types and to organize large-scale programs and systems
- The standard library, which uses these advanced features to provide useful data structures and algorithms

Most texts present C++ in the order in which it evolved. They teach the C subset of C++ first, and present the more abstract features of C++ as advanced topics at the end of the book. There are two problems with this approach: Readers can get bogged down in the details inherent in low-level programming and give up in frustration. Those who do press on learn bad habits that they must unlearn later.

We take the opposite approach: Right from the start, we use the features that let programmers ignore the details inherent in low-level programming. For example, we introduce and use the library `string` and `vector` types along with the built-in arithmetic and array types. Programs that use these library types are easier to write, easier to understand, and much less error-prone.

Too often, the library is taught as an “advanced” topic. Instead of using the library, many books use low-level programming techniques based on pointers to character arrays and dynamic memory management. Getting programs that use these low-level techniques to work correctly is much harder than writing the corresponding C++ code using the library.

Throughout *C++ Primer*, we emphasize good style: We want to help you, the reader, develop good habits immediately and avoid needing to unlearn bad habits as you gain more sophisticated knowledge. We highlight particularly tricky matters and warn about common misconceptions and pitfalls.

We also explain the rationale behind the rules—explaining the why not just the what. We believe that by understanding why things work as they do, readers can more quickly cement their grasp of the language.

Although you do not need to know C in order to understand this book, we assume you know enough about programming to write, compile, and run a program in at least one modern block-structured language. In particular, we assume you have used variables, written and called functions, and used a compiler.

## Changes to the Fifth Edition

New to this edition of *C++ Primer* are icons in the margins to help guide the reader. C++ is a large language that offers capabilities tailored to particular kinds of programming problems. Some of these capabilities are of great import for large project teams but might not be necessary for smaller efforts. As a result, not every programmer needs to know every detail of every feature. We’ve added these marginal icons to help the reader know which parts can be learned later and which topics are more essential.



We’ve marked sections that cover the fundamentals of the language with an image of a person studying a book. The topics covered in sections marked this

way form the core part of the language. Everyone should read and understand these sections.

We've also indicated those sections that cover advanced or special-purpose topics. These sections can be skipped or skimmed on a first reading. We've marked such sections with a stack of books to indicate that you can safely put down the book at that point. It is probably a good idea to skim such sections so you know that the capability exists. However, there is no reason to spend time studying these topics until you actually need to use the feature in your own programs.



To help readers guide their attention further, we've noted particularly tricky concepts with a magnifying-glass icon. We hope that readers will take the time to understand thoroughly the material presented in the sections so marked. In at least some of these sections, the import of the topic may not be readily apparent; but we think you'll find that these sections cover topics that turn out to be essential to understanding the language.



Another aid to reading this book, is our extensive use of cross-references. We hope these references will make it easier for readers to dip into the middle of the book, yet easily jump back to the earlier material on which later examples rely.

What remains unchanged is that *C++ Primer* is a clear, correct, and thorough tutorial guide to C++. We teach the language by presenting a series of increasingly sophisticated examples, which explain language features and show how to make the best use of C++.

## Structure of This Book

We start by covering the basics of the language and the library together in Parts I and II. These parts cover enough material to let you, the reader, write significant programs. Most C++ programmers need to know essentially everything covered in this portion of the book.

In addition to teaching the basics of C++, the material in Parts I and II serves another important purpose: By using the abstract facilities defined by the library, you will become more comfortable with using high-level programming techniques. The library facilities are themselves abstract data types that are usually written in C++. The library can be defined using the same class-construction features that are available to any C++ programmer. Our experience in teaching C++ is that by first using well-designed abstract types, readers find it easier to understand how to build their own types.

Only after a thorough grounding in using the library—and writing the kinds of abstract programs that the library allows—do we move on to those C++ features that will enable you to write your own abstractions. Parts III and IV focus on writing abstractions in the form of classes. Part III covers the fundamentals; Part IV covers more specialized facilities.

In Part III, we cover issues of copy control, along with other techniques to make classes that are as easy to use as the built-in types. Classes are the foundation for object-oriented and generic programming, which we also cover in Part III. *C++ Primer* concludes with Part IV, which covers features that are of most use in structuring large, complicated systems. We also summarize the library algorithms in Appendix A.

## Aids to the Reader

Each chapter concludes with a summary, followed by a glossary of defined terms, which together recap the chapter's most important points. Readers should use these sections as a personal checklist: If you do not understand a term, restudy the corresponding part of the chapter.

We've also incorporated a number of other learning aids in the body of the text:

- Important terms are indicated in **bold**; important terms that we assume are already familiar to the reader are indicated in *bold italics*. Each term appears in the chapter's Defined Terms section.
- Throughout the book, we highlight parts of the text to call attention to important aspects of the language, warn about common pitfalls, suggest good programming practices, and provide general usage tips.
- To make it easier to follow the relationships among features and concepts, we provide extensive forward and backward cross-references.
- We provide sidebar discussions on important concepts and for topics that new C++ programmers often find most difficult.
- Learning any programming language requires writing programs. To that end, the Primer provides extensive examples throughout the text. Source code for the extended examples is available on the Web at the following URL:

<http://www.informit.com/title/0321714113>

## A Note about Compilers

As of this writing (July, 2012), compiler vendors are hard at work updating their compilers to match the latest ISO standard. The compiler we use most frequently is the GNU compiler, version 4.7.0. There are only a few features used in this book that this compiler does not yet implement: inheriting constructors, reference qualifiers for member functions, and the regular-expression library.

## Acknowledgments

In preparing this edition we are very grateful for the help of several current and former members of the standardization committee: Dave Abrahams, Andy Koenig, Stephan T. Lavavej, Jason Merrill, John Spicer, and Herb Sutter. They provided invaluable assistance to us in understanding some of the more subtle parts of the new standard. We'd also like to thank the many folks who worked on updating the GNU compiler making the standard a reality.

As in previous editions of C++ *Primer*, we'd like to extend our thanks to Bjarne Stroustrup for his tireless work on C++ and for his friendship to the authors during most of that time. We'd also like to thank Alex Stepanov for his original insights that led to the containers and algorithms at the core of the standard library. Finally, our thanks go to all the C++ Standards committee members for their hard work in clarifying, refining, and improving C++ over many years.

We extend our deep-felt thanks to our reviewers, whose helpful comments led us to make improvements great and small throughout the book: Marshall Clow, Jon Kalb, Nevin Liber, Dr. C. L. Tondo, Daveed Vandevoorde, and Steve Vinoski.

This book was typeset using  $\text{\LaTeX}$  and the many packages that accompany the  $\text{\LaTeX}$  distribution. Our well-justified thanks go to the members of the  $\text{\LaTeX}$  community, who have made available such powerful typesetting tools.

Finally, we thank the fine folks at Addison-Wesley who have shepherded this edition through the publishing process: Peter Gordon, our editor, who provided the impetus for us to revise *C++ Primer* once again; Kim Boedigheimer, who keeps us all on schedule; Barbara Wood, who found lots of editing errors for us during the copy-edit phase, and Elizabeth Ryan, who was again a delight to work with as she guided us through the design and production process.

*This page intentionally left blank*

C H A P T E R

1

GETTING STARTED

CONTENTS

---

Section 1.1	Writing a Simple C++ Program . . . . .	2
Section 1.2	A First Look at Input/Output . . . . .	5
Section 1.3	A Word about Comments . . . . .	9
Section 1.4	Flow of Control . . . . .	11
Section 1.5	Introducing Classes . . . . .	19
Section 1.6	The Bookstore Program . . . . .	24
Chapter Summary	. . . . .	26
Defined Terms	. . . . .	26

This chapter introduces most of the basic elements of C++: types, variables, expressions, statements, and functions. Along the way, we'll briefly explain how to compile and execute a program.

After having read this chapter and worked through the exercises, you should be able to write, compile, and execute simple programs. Later chapters will assume that you can use the features introduced in this chapter, and will explain these features in more detail.

*The way to learn* a new programming language is to write programs. In this chapter, we'll write a program to solve a simple problem for a bookstore.

Our store keeps a file of transactions, each of which records the sale of one or more copies of a single book. Each transaction contains three data elements:

0-201-70353-X 4 24.99

The first element is an ISBN (International Standard Book Number, a unique book identifier), the second is the number of copies sold, and the last is the price at which each of these copies was sold. From time to time, the bookstore owner reads this file and for each book computes the number of copies sold, the total revenue from that book, and the average sales price.

To be able to write this program, we need to cover a few basic C++ features. In addition, we'll need to know how to compile and execute a program.

Although we haven't yet designed our program, it's easy to see that it must

- Define variables
- Do input and output
- Use a data structure to hold the data
- Test whether two records have the same ISBN
- Contain a loop that will process every record in the transaction file

We'll start by reviewing how to solve these subproblems in C++ and then write our bookstore program.

## 1.1 Writing a Simple C++ Program

Every C++ program contains one or more *functions*, one of which must be named **main**. The operating system runs a C++ program by calling **main**. Here is a simple version of **main** that does nothing but return a value to the operating system:

```
int main()
{
    return 0;
}
```

A function definition has four elements: a *return type*, a *function name*, a (possibly empty) *parameter list* enclosed in parentheses, and a *function body*. Although **main** is special in some ways, we define **main** the same way we define any other function.

In this example, **main** has an empty list of parameters (shown by the `()` with nothing inside). § 6.2.5 (p. 218) will discuss the other parameter types that we can define for **main**.

The **main** function is required to have a return type of **int**, which is a type that represents integers. The **int** type is a **built-in type**, which means that it is one of the types the language defines.

The final part of a function definition, the function body, is a *block of statements* starting with an open **curly brace** and ending with a close curly:

```
{  
    return 0;  
}
```

The only statement in this block is a `return`, which is a statement that terminates a function. As is the case here, a `return` can also send a value back to the function's caller. When a `return` statement includes a value, the value returned must have a type that is compatible with the return type of the function. In this case, the return type of `main` is `int` and the return value is `0`, which is an `int`.



Note the semicolon at the end of the `return` statement. Semicolons mark the end of most statements in C++. They are easy to overlook but, when forgotten, can lead to mysterious compiler error messages.

On most systems, the value returned from `main` is a status indicator. A return value of `0` indicates success. A nonzero return has a meaning that is defined by the system. Ordinarily a nonzero return indicates what kind of error occurred.

#### KEY CONCEPT: TYPES

Types are one of the most fundamental concepts in programming and a concept that we will come back to over and over in this Primer. A type defines both the contents of a data element and the operations that are possible on those data.

The data our programs manipulate are stored in variables and every variable has a type. When the type of a variable named `v` is `T`, we often say that “`v` has type `T`” or, interchangeably, that “`v` is a `T`.”

### 1.1.1 Compiling and Executing Our Program

Having written the program, we need to compile it. How you compile a program depends on your operating system and compiler. For details on how your particular compiler works, check the reference manual or ask a knowledgeable colleague.

Many PC-based compilers are run from an integrated development environment (IDE) that bundles the compiler with build and analysis tools. These environments can be a great asset in developing large programs but require a fair bit of time to learn how to use effectively. Learning how to use such environments is well beyond the scope of this book.

Most compilers, including those that come with an IDE, provide a command-line interface. Unless you already know the IDE, you may find it easier to start with the command-line interface. Doing so will let you concentrate on learning C++ first. Moreover, once you understand the language, the IDE is likely to be easier to learn.

#### Program Source File Naming Convention

Whether you use a command-line interface or an IDE, most compilers expect program source code to be stored in one or more files. Program files are normally



referred to as a *source files*. On most systems, the name of a source file ends with a suffix, which is a period followed by one or more characters. The suffix tells the system that the file is a C++ program. Different compilers use different suffix conventions; the most common include `.cc`, `.cxx`, `.cpp`, `.cp`, and `.C`.

## Running the Compiler from the Command Line

If we are using a command-line interface, we will typically compile a program in a console window (such as a shell window on a UNIX system or a Command Prompt window on Windows). Assuming that our main program is in a file named `prog1.cc`, we might compile it by using a command such as

```
$ CC prog1.cc
```

where `CC` names the compiler and `$` is the system prompt. The compiler generates an executable file. On a Windows system, that executable file is named `prog1.exe`. UNIX compilers tend to put their executables in files named `a.out`.

To run an executable on Windows, we supply the executable file name and can omit the `.exe` file extension:

```
$ prog1
```

On some systems you must specify the file's location explicitly, even if the file is in the current directory or folder. In such cases, we would write

```
$ .\prog1
```

The `"."` followed by a backslash indicates that the file is in the current directory.

To run an executable on UNIX, we use the full file name, including the file extension:

```
$ a.out
```

If we need to specify the file's location, we'd use a `"."` followed by a forward slash to indicate that our executable is in the current directory:

```
$ ./a.out
```

The value returned from `main` is accessed in a system-dependent manner. On both UNIX and Windows systems, after executing the program, you must issue an appropriate `echo` command.

On UNIX systems, we obtain the status by writing

```
$ echo $?
```

To see the status on a Windows system, we write

```
$ echo %ERRORLEVEL%
```

### RUNNING THE GNU OR MICROSOFT COMPILERS

The command used to run the C++ compiler varies across compilers and operating systems. The most common compilers are the GNU compiler and the Microsoft Visual Studio compilers. By default, the command to run the GNU compiler is `g++`:

```
$ g++ -o prog1 prog1.cc
```

Here `$` is the system prompt. The `-o prog1` is an argument to the compiler and names the file in which to put the executable file. This command generates an executable file named `prog1` or `prog1.exe`, depending on the operating system. On UNIX, executable files have no suffix; on Windows, the suffix is `.exe`. If the `-o prog1` is omitted, the compiler generates an executable named `a.out` on UNIX systems and `a.exe` on Windows. (Note: Depending on the release of the GNU compiler you are using, you may need to specify `-std=c++0x` to turn on C++ 11 support.)

The command to run the Microsoft Visual Studio 2010 compiler is `cl`:

```
C:\Users\me\Programs> cl /EHsc prog1.cpp
```

Here `C:\Users\me\Programs>` is the system prompt and `\Users\me\Programs` is the name of the current directory (aka the current folder). The `cl` command invokes the compiler, and `/EHsc` is the compiler option that turns on standard exception handling. The Microsoft compiler automatically generates an executable with a name that corresponds to the first source file name. The executable has the suffix `.exe` and the same name as the source file name. In this case, the executable is named `prog1.exe`.

Compilers usually include options to generate warnings about problematic constructs. It is usually a good idea to use these options. Our preference is to use `-Wall` with the GNU compiler, and to use `/W4` with the Microsoft compilers.

For further information consult your compiler's user's guide.

### EXERCISES SECTION 1.1.1

**Exercise 1.1:** Review the documentation for your compiler and determine what file naming convention it uses. Compile and run the `main` program from page 2.

**Exercise 1.2:** Change the program to return `-1`. A return value of `-1` is often treated as an indicator that the program failed. Recompile and rerun your program to see how your system treats a failure indicator from `main`.

## 1.2 A First Look at Input/Output

The C++ language does not define any statements to do input or output (IO). Instead, C++ includes an extensive **standard library** that provides IO (and many other facilities). For many purposes, including the examples in this book, one needs to know only a few basic concepts and operations from the IO library.

Most of the examples in this book use the **`iostream`** library. Fundamental to the **`iostream`** library are two types named **`istream`** and **`ostream`**, which represent input and output streams, respectively. A stream is a sequence of characters read from or written to an IO device. The term *stream* is intended to suggest that the characters are generated, or consumed, sequentially over time.

## Standard Input and Output Objects

The library defines four IO objects. To handle input, we use an object of type `istream` named `cin` (pronounced *see-in*). This object is also referred to as the **standard input**. For output, we use an `ostream` object named `cout` (pronounced *see-out*). This object is also known as the **standard output**. The library also defines two other `ostream` objects, named `cerr` and `clog` (pronounced *see-err* and *see-log*, respectively). We typically use `cerr`, referred to as the **standard error**, for warning and error messages and `clog` for general information about the execution of the program.

Ordinarily, the system associates each of these objects with the window in which the program is executed. So, when we read from `cin`, data are read from the window in which the program is executing, and when we write to `cout`, `cerr`, or `clog`, the output is written to the same window.

## A Program That Uses the IO Library

In our bookstore problem, we'll have several records that we'll want to combine into a single total. As a simpler, related problem, let's look first at how we might add two numbers. Using the IO library, we can extend our main program to prompt the user to give us two numbers and then print their sum:

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

This program starts by printing

```
Enter two numbers:
```

on the user's screen and then waits for input from the user. If the user enters

```
3 7
```

followed by a newline, then the program produces the following output:

```
The sum of 3 and 7 is 10
```

The first line of our program

```
#include <iostream>
```

tells the compiler that we want to use the `iostream` library. The name inside angle brackets (`iostream` in this case) refers to a **header**. Every program that uses a library facility must include its associated header. The `#include` directive

must be written on a single line—the name of the header and the `#include` must appear on the same line. In general, `#include` directives must appear outside any function. Typically, we put all the `#include` directives for a program at the beginning of the source file.

## Writing to a Stream

The first statement in the body of `main` executes an **expression**. In C++ an expression yields a result and is composed of one or more operands and (usually) an operator. The expressions in this statement use the output operator (the **« operator**) to print a message on the standard output:

```
std::cout << "Enter two numbers:" << std::endl;
```

The `<<` operator takes two operands: The left-hand operand must be an `ostream` object; the right-hand operand is a value to print. The operator writes the given value on the given `ostream`. The result of the output operator is its left-hand operand. That is, the result is the `ostream` on which we wrote the given value.

Our output statement uses the `<<` operator twice. Because the operator returns its left-hand operand, the result of the first operator becomes the left-hand operand of the second. As a result, we can chain together output requests. Thus, our expression is equivalent to

```
(std::cout << "Enter two numbers:") << std::endl;
```

Each operator in the chain has the same object as its left-hand operand, in this case `std::cout`. Alternatively, we can generate the same output using two statements:

```
std::cout << "Enter two numbers:";  
std::cout << std::endl;
```

The first output operator prints a message to the user. That message is a **string literal**, which is a sequence of characters enclosed in double quotation marks. The text between the quotation marks is printed to the standard output.

The second operator prints `endl`, which is a special value called a **manipulator**. Writing `endl` has the effect of ending the current line and flushing the *buffer* associated with that device. Flushing the buffer ensures that all the output the program has generated so far is actually written to the output stream, rather than sitting in memory waiting to be written.



Programmers often add print statements during debugging. Such statements should *always* flush the stream. Otherwise, if the program crashes, output may be left in the buffer, leading to incorrect inferences about where the program crashed.

## Using Names from the Standard Library

Careful readers will note that this program uses `std::cout` and `std::endl` rather than just `cout` and `endl`. The prefix `std::` indicates that the names `cout` and `endl` are defined inside the **namespace** named `std`. Namespaces allow us to

avoid inadvertent collisions between the names we define and uses of those same names inside a library. All the names defined by the standard library are in the `std` namespace.

One side effect of the library's use of a namespace is that when we use a name from the library, we must say explicitly that we want to use the name from the `std` namespace. Writing `std::cout` uses the scope operator (the `::` **operator**) to say that we want to use the name `cout` that is defined in the namespace `std`. § 3.1 (p. 82) will show a simpler way to access names from the library.

## Reading from a Stream

Having asked the user for input, we next want to read that input. We start by defining two *variables* named `v1` and `v2` to hold the input:

```
int v1 = 0, v2 = 0;
```

We define these variables as type `int`, which is a built-in type representing integers. We also *initialize* them to 0. When we initialize a variable, we give it the indicated value at the same time as the variable is created.

The next statement

```
std::cin >> v1 >> v2;
```

reads the input. The input operator (the `>>` **operator**) behaves analogously to the output operator. It takes an `istream` as its left-hand operand and an object as its right-hand operand. It reads data from the given `istream` and stores what was read in the given object. Like the output operator, the input operator returns its left-hand operand as its result. Hence, this expression is equivalent to

```
(std::cin >> v1) >> v2;
```

Because the operator returns its left-hand operand, we can combine a sequence of input requests into a single statement. Our input operation reads two values from `std::cin`, storing the first in `v1` and the second in `v2`. In other words, our input operation executes as

```
std::cin >> v1;
std::cin >> v2;
```

## Completing the Program

What remains is to print our result:

```
std::cout << "The sum of " << v1 << " and " << v2
          << " is " << v1 + v2 << std::endl;
```

This statement, although longer than the one that prompted the user for input, is conceptually similar. It prints each of its operands on the standard output. What is interesting in this example is that the operands are not all the same kinds of values. Some operands are string literals, such as `"The sum of "`. Others are `int` values, such as `v1`, `v2`, and the result of evaluating the arithmetic expression `v1 + v2`. The library defines versions of the input and output operators that handle operands of each of these differing types.

**EXERCISES SECTION 1.2**

**Exercise 1.3:** Write a program to print `Hello, World` on the standard output.

**Exercise 1.4:** Our program used the addition operator, `+`, to add two numbers. Write a program that uses the multiplication operator, `*`, to print the product instead.

**Exercise 1.5:** We wrote the output in one large statement. Rewrite the program to use a separate statement to print each operand.

**Exercise 1.6:** Explain whether the following program fragment is legal.

```
std::cout << "The sum of " << v1;
          << " and " << v2;
          << " is " << v1 + v2 << std::endl;
```

If the program is legal, what does it do? If the program is not legal, why not? How would you fix it?

## 1.3 A Word about Comments

Before our programs get much more complicated, we should see how C++ handles *comments*. Comments help the human readers of our programs. They are typically used to summarize an algorithm, identify the purpose of a variable, or clarify an otherwise obscure segment of code. The compiler ignores comments, so they have no effect on the program's behavior or performance.

Although the compiler ignores comments, readers of our code do not. Programmers tend to believe comments even when other parts of the system documentation are out of date. An incorrect comment is worse than no comment at all because it may mislead the reader. When you change your code, be sure to update the comments, too!

### Kinds of Comments in C++

There are two kinds of comments in C++: single-line and paired. A single-line comment starts with a double slash (`//`) and ends with a newline. Everything to the right of the slashes on the current line is ignored by the compiler. A comment of this kind can contain any text, including additional double slashes.

The other kind of comment uses two delimiters (`/*` and `*/`) that are inherited from C. Such comments begin with a `/*` and end with the next `*/`. These comments can include anything that is not a `*/`, including newlines. The compiler treats everything that falls between the `/*` and `*/` as part of the comment.

A comment pair can be placed anywhere a tab, space, or newline is permitted. Comment pairs can span multiple lines of a program but are not required to do so. When a comment pair does span multiple lines, it is often a good idea to indicate visually that the inner lines are part of a multiline comment. Our style is to begin each line in the comment with an asterisk, thus indicating that the entire range is part of a multiline comment.

Programs typically contain a mixture of both comment forms. Comment pairs

generally are used for multiline explanations, whereas double-slash comments tend to be used for half-line and single-line remarks:

```
#include <iostream>

/*
 * Simple main function:
 * Read two numbers and write their sum
 */
int main()
{
    // prompt user to enter two numbers
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;    // variables to hold the input we read
    std::cin >> v1 >> v2; // read input
    std::cout << "The sum of " << v1 << " and " << v2
                << " is " << v1 + v2 << std::endl;
    return 0;
}
```



In this book, we italicize comments to make them stand out from the normal program text. In actual programs, whether comment text is distinguished from the text used for program code depends on the sophistication of the programming environment you are using.

## Comment Pairs Do Not Nest

A comment that begins with `/*` ends with the next `*/`. As a result, one comment pair cannot appear inside another. The compiler error messages that result from this kind of mistake can be mysterious and confusing. As an example, compile the following program on your system:

```
/*
 * comment pairs /* */ cannot nest.
 * "cannot nest" is considered source code,
 * as is the rest of the program
 */
int main()
{
    return 0;
}
```

We often need to comment out a block of code during debugging. Because that code might contain nested comment pairs, the best way to comment a block of code is to insert single-line comments at the beginning of each line in the section we want to ignore:

```
// /*
// * everything inside a single-line comment is ignored
// * including nested comment pairs
// */
```

**EXERCISES SECTION 1.3**

**Exercise 1.7:** Compile a program that has incorrectly nested comments.

**Exercise 1.8:** Indicate which, if any, of the following output statements are legal:

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
std::cout << /* "*/" /* "/*" */;
```

After you've predicted what will happen, test your answers by compiling a program with each of these statements. Correct any errors you encounter.

## 1.4 Flow of Control

Statements normally execute sequentially: The first statement in a block is executed first, followed by the second, and so on. Of course, few programs—including the one to solve our bookstore problem—can be written using only sequential execution. Instead, programming languages provide various flow-of-control statements that allow for more complicated execution paths.

### 1.4.1 The `while` Statement

A **while statement** repeatedly executes a section of code so long as a given condition is true. We can use a `while` to write a program to sum the numbers from 1 through 10 inclusive as follows:

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // keep executing the while as long as val is less than or equal to 10
    while (val <= 10) {
        sum += val; // assigns sum + val to sum
        ++val;     // add 1 to val
    }
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

When we compile and execute this program, it prints

```
Sum of 1 to 10 inclusive is 55
```

As before, we start by including the `iostream` header and defining `main`. Inside `main` we define two `int` variables: `sum`, which will hold our summation, and `val`, which will represent each of the values from 1 through 10. We give `sum` an initial value of 0 and start `val` off with the value 1.



The new part of this program is the `while` statement. A `while` has the form

```
while (condition)
    statement
```

A `while` executes by (alternately) testing the *condition* and executing the associated *statement* until the *condition* is false. A **condition** is an expression that yields a result that is either true or false. So long as *condition* is true, *statement* is executed. After executing *statement*, *condition* is tested again. If *condition* is again true, then *statement* is again executed. The `while` continues, alternately testing the *condition* and executing *statement* until the *condition* is false.

In this program, the `while` statement is

```
// keep executing the while as long as val is less than or equal to 10
while (val <= 10) {
    sum += val; // assigns sum + val to sum
    ++val;     // add 1 to val
}
```

The condition uses the less-than-or-equal operator (the **<= operator**) to compare the current value of `val` and 10. As long as `val` is less than or equal to 10, the condition is true. If the condition is true, we execute the body of the `while`. In this case, that body is a block with two statements:

```
{
    sum += val; // assigns sum + val to sum
    ++val;     // add 1 to val
}
```

A block is a sequence of zero or more statements enclosed by curly braces. A block is a statement and may be used wherever a statement is required. The first statement in this block uses the compound assignment operator (the **+= operator**). This operator adds its right-hand operand to its left-hand operand and stores the result in the left-hand operand. It has essentially the same effect as writing an addition and an **assignment**:

```
sum = sum + val; // assign sum + val to sum
```

Thus, the first statement in the block adds the value of `val` to the current value of `sum` and stores the result back into `sum`.

The next statement

```
++val; // add 1 to val
```

uses the prefix increment operator (the **++ operator**). The increment operator adds 1 to its operand. Writing `++val` is the same as writing `val = val + 1`.

After executing the `while` body, the loop evaluates the condition again. If the (now incremented) value of `val` is still less than or equal to 10, then the body of the `while` is executed again. The loop continues, testing the condition and executing the body, until `val` is no longer less than or equal to 10.

Once `val` is greater than 10, the program falls out of the `while` loop and continues execution with the statement following the `while`. In this case, that statement prints our output, followed by the `return`, which completes our main program.

**EXERCISES SECTION 1.4.1**

**Exercise 1.9:** Write a program that uses a `while` to sum the numbers from 50 to 100.

**Exercise 1.10:** In addition to the `++` operator that adds 1 to its operand, there is a decrement operator (`--`) that subtracts 1. Use the decrement operator to write a `while` that prints the numbers from ten down to zero.

**Exercise 1.11:** Write a program that prompts the user for two integers. Print each number in the range specified by those two integers.

**1.4.2 The for Statement**

In our `while` loop we used the variable `val` to control how many times we executed the loop. We tested the value of `val` in the condition and incremented `val` in the `while` body.

This pattern—using a variable in a condition and incrementing that variable in the body—happens so often that the language defines a second statement, the **for statement**, that abbreviates code that follows this pattern. We can rewrite this program using a `for` loop to sum the numbers from 1 through 10 as follows:

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
               << sum << std::endl;
    return 0;
}
```

As before, we define `sum` and initialize it to zero. In this version, we define `val` as part of the `for` statement itself:

```
for (int val = 1; val <= 10; ++val)
    sum += val;
```

Each `for` statement has two parts: a header and a body. The header controls how often the body is executed. The header itself consists of three parts: an *init-statement*, a *condition*, and an *expression*. In this case, the *init-statement*

```
int val = 1;
```

defines an `int` object named `val` and gives it an initial value of 1. The variable `val` exists only inside the `for`; it is not possible to use `val` after this loop terminates. The *init-statement* is executed only once, on entry to the `for`. The *condition*

```
val <= 10
```

compares the current value in `val` to 10. The *condition* is tested each time through the loop. As long as `val` is less than or equal to 10, we execute the `for` body. The *expression* is executed after the `for` body. Here, the *expression*

```
++val
```

uses the prefix increment operator, which adds 1 to the value of `val`. After executing the *expression*, the `for` retests the *condition*. If the new value of `val` is still less than or equal to 10, then the `for` loop body is executed again. After executing the body, `val` is incremented again. The loop continues until the *condition* fails.

In this loop, the `for` body performs the summation

```
sum += val; // equivalent to sum = sum + val
```

To recap, the overall execution flow of this `for` is:

1. Create `val` and initialize it to 1.
2. Test whether `val` is less than or equal to 10. If the test succeeds, execute the `for` body. If the test fails, exit the loop and continue execution with the first statement following the `for` body.
3. Increment `val`.
4. Repeat the test in step 2, continuing with the remaining steps as long as the condition is true.

## EXERCISES SECTION 1.4.2

**Exercise 1.12:** What does the following `for` loop do? What is the final value of `sum`?

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

**Exercise 1.13:** Rewrite the first two exercises from § 1.4.1 (p. 13) using `for` loops.

**Exercise 1.14:** Compare and contrast the loops that used a `for` with those using a `while`. Are there advantages or disadvantages to using either form?

**Exercise 1.15:** Write programs that contain the common errors discussed in the box on page 16. Familiarize yourself with the messages the compiler generates.

## 1.4.3 Reading an Unknown Number of Inputs

In the preceding sections, we wrote programs that summed the numbers from 1 through 10. A logical extension of this program would be to ask the user to input a set of numbers to sum. In this case, we won't know how many numbers to add. Instead, we'll keep reading numbers until there are no more numbers to read:

```
#include <iostream>
int main()
{
    int sum = 0, value = 0;
    // read until end-of-file, calculating a running total of all values read
    while (std::cin >> value)
        sum += value; // equivalent to sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

If we give this program the input

**3 4 5 6**

then our output will be

**Sum is: 18**

The first line inside `main` defines two `int` variables, named `sum` and `value`, which we initialize to 0. We'll use `value` to hold each number as we read it from the input. We read the data inside the condition of the `while`:

```
while (std::cin >> value)
```

Evaluating the `while` condition executes the expression

```
std::cin >> value
```

That expression reads the next number from the standard input and stores that number in `value`. The input operator (§ 1.2, p. 8) returns its left operand, which in this case is `std::cin`. This condition, therefore, tests `std::cin`.

When we use an `istream` as a condition, the effect is to test the state of the stream. If the stream is valid—that is, if the stream hasn't encountered an error—then the test succeeds. An `istream` becomes invalid when we hit *end-of-file* or encounter an invalid input, such as reading a value that is not an integer. An `istream` that is in an invalid state will cause the condition to yield false.

Thus, our `while` executes until we encounter end-of-file (or an input error). The `while` body uses the compound assignment operator to add the current value to the evolving sum. Once the condition fails, the `while` ends. We fall through and execute the next statement, which prints the sum followed by `endl`.

#### ENTERING AN END-OF-FILE FROM THE KEYBOARD

When we enter input to a program from the keyboard, different operating systems use different conventions to allow us to indicate end-of-file. On Windows systems we enter an end-of-file by typing a control-z—hold down the Ctrl key and press z—followed by hitting either the Enter or Return key. On UNIX systems, including on Mac OS X machines, end-of-file is usually control-d.

## COMPILATION REVISITED

Part of the compiler's job is to look for errors in the program text. A compiler cannot detect whether a program does what its author intends, but it can detect errors in the *form* of the program. The following are the most common kinds of errors a compiler will detect.

**Syntax errors:** The programmer has made a grammatical error in the C++ language. The following program illustrates common syntax errors; each comment describes the error on the following line:

```
// error: missing ) in parameter list for main
int main ( {
    // error: used colon, not a semicolon, after endl
    std::cout << "Read each file." << std::endl:
    // error: missing quotes around string literal
    std::cout << Update master. << std::endl;
    // error: second output operator is missing
    std::cout << "Write new master." std::endl;
    // error: missing ; on return statement
    return 0
}
```

**Type errors:** Each item of data in C++ has an associated type. The value 10, for example, has a type of `int` (or, more colloquially, "is an `int`"). The word "hello", including the double quotation marks, is a string literal. One example of a type error is passing a string literal to a function that expects an `int` argument.

**Declaration errors:** Every name used in a C++ program must be declared before it is used. Failure to declare a name usually results in an error message. The two most common declaration errors are forgetting to use `std::` for a name from the library and misspelling the name of an identifier:

```
#include <iostream>

int main()
{
    int v1 = 0, v2 = 0;
    std::cin >> v >> v2; // error: uses "v" not "v1"
    // error: cout not defined; should be std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
```

Error messages usually contain a line number and a brief description of what the compiler believes we have done wrong. It is a good practice to correct errors in the sequence they are reported. Often a single error can have a cascading effect and cause a compiler to report more errors than actually are present. It is also a good idea to recompile the code after each fix—or after making at most a small number of obvious fixes. This cycle is known as *edit-compile-debug*.

**EXERCISES SECTION 1.4.3**

**Exercise 1.16:** Write your own version of a program that prints the sum of a set of integers read from `cin`.

**1.4.4 The `if` Statement**

Like most languages, C++ provides an **`if` statement** that supports conditional execution. We can use an `if` to write a program to count how many consecutive times each distinct value appears in the input:

```
#include <iostream>

int main()
{
    // currVal is the number we're counting; we'll read new values into val
    int currVal = 0, val = 0;
    // read first number and ensure that we have data to process
    if (std::cin >> currVal) {
        int cnt = 1; // store the count for the current value we're processing
        while (std::cin >> val) { // read the remaining numbers
            if (val == currVal) // if the values are the same
                ++cnt; // add 1 to cnt
            else { // otherwise, print the count for the previous value
                std::cout << currVal << " occurs "
                    << cnt << " times" << std::endl;
                currVal = val; // remember the new value
                cnt = 1; // reset the counter
            }
        } // while loop ends here
        // remember to print the count for the last value in the file
        std::cout << currVal << " occurs "
            << cnt << " times" << std::endl;
    } // outermost if statement ends here
    return 0;
}
```

If we give this program the following input:

```
42 42 42 42 42 55 55 62 100 100 100
```

then the output should be

```
42 occurs 5 times
55 occurs 2 times
62 occurs 1 times
100 occurs 3 times
```

Much of the code in this program should be familiar from our earlier programs. We start by defining `val` and `currVal`: `currVal` will keep track of which number we are counting; `val` will hold each number as we read it from the input. What's new are the two `if` statements. The first `if`

```

if (std::cin >> currVal) {
    // ...
} // outermost if statement ends here

```

ensures that the input is not empty. Like a `while`, an `if` evaluates a condition. The condition in the first `if` reads a value into `currVal`. If the read succeeds, then the condition is true and we execute the block that starts with the open curly following the condition. That block ends with the close curly just before the `return` statement.

Once we know there are numbers to count, we define `cnt`, which will count how often each distinct number occurs. We use a `while` loop similar to the one in the previous section to (repeatedly) read numbers from the standard input.

The body of the `while` is a block that contains the second `if` statement:

```

if (val == currVal)    // if the values are the same
    ++cnt;             // add 1 to cnt
else { // otherwise, print the count for the previous value
    std::cout << currVal << " occurs "
               << cnt << " times" << std::endl;
    currVal = val;     // remember the new value
    cnt = 1;          // reset the counter
}

```

The condition in this `if` uses the equality operator (the **`==` operator**) to test whether `val` is equal to `currVal`. If so, we execute the statement that immediately follows the condition. That statement increments `cnt`, indicating that we have seen `currVal` once more.

If the condition is false—that is, if `val` is not equal to `currVal`—then we execute the statement following the `else`. This statement is a block consisting of an output statement and two assignments. The output statement prints the count for the value we just finished processing. The assignments reset `cnt` to 1 and `currVal` to `val`, which is the number we just read.



**WARNING**

C++ uses `=` for assignment and `==` for equality. Both operators can appear inside a condition. It is a common mistake to write `=` when you mean `==` inside a condition.

## EXERCISES SECTION 1.4.4

**Exercise 1.17:** What happens in the program presented in this section if the input values are all equal? What if there are no duplicated values?

**Exercise 1.18:** Compile and run the program from this section giving it only equal values as input. Run it again giving it values in which no number is repeated.

**Exercise 1.19:** Revise the program you wrote for the exercises in § 1.4.1 (p. 13) that printed a range of numbers so that it handles input in which the first number is smaller than the second.

**KEY CONCEPT: INDENTATION AND FORMATTING OF C++ PROGRAMS**

C++ programs are largely free-format, meaning that where we put curly braces, indentation, comments, and newlines usually has no effect on what our programs mean. For example, the curly brace that denotes the beginning of the body of `main` could be on the same line as `main`; positioned as we have done, at the beginning of the next line; or placed anywhere else we'd like. The only requirement is that the open curly must be the first nonblank, noncomment character following `main`'s parameter list.

Although we are largely free to format programs as we wish, the choices we make affect the readability of our programs. We could, for example, have written `main` on a single long line. Such a definition, although legal, would be hard to read.

Endless debates occur as to the right way to format C or C++ programs. Our belief is that there is no single correct style but that there is value in consistency. Most programmers indent subsidiary parts of their programs, as we've done with the statements inside `main` and the bodies of our loops. We tend to put the curly braces that delimit functions on their own lines. We also indent compound IO expressions so that the operators line up. Other indentation conventions will become clear as our programs become more sophisticated.

The important thing to keep in mind is that other ways to format programs are possible. When you choose a formatting style, think about how it affects readability and comprehension. Once you've chosen a style, use it consistently.

## 1.5 Introducing Classes

The only remaining feature we need to understand before solving our bookstore problem is how to define a *data structure* to represent our transaction data. In C++ we define our own data structures by defining a **class**. A class defines a type along with a collection of operations that are related to that type. The class mechanism is one of the most important features in C++. In fact, a primary focus of the design of C++ is to make it possible to define **class types** that behave as naturally as the built-in types.

In this section, we'll describe a simple class that we can use in writing our bookstore program. We'll implement this class in later chapters as we learn more about types, expressions, statements, and functions.

To use a class we need to know three things:

- What is its name?
- Where is it defined?
- What operations does it support?

For our bookstore problem, we'll assume that the class is named `Sales_item` and that it is already defined in a header named `Sales_item.h`.

As we've seen, to use a library facility, we must include the associated header. Similarly, we use headers to access classes defined for our own applications. Conventionally, header file names are derived from the name of a class defined in that header. Header files that we write usually have a suffix of `.h`, but some programmers use `.H`, `.hpp`, or `.hxx`. The standard library headers typically have no suffix



at all. Compilers usually don't care about the form of header file names, but IDEs sometimes do.

## 1.5.1 The `Sales_item` Class

The purpose of the `Sales_item` class is to represent the total revenue, number of copies sold, and average sales price for a book. How these data are stored or computed is not our concern. To use a class, we need not care about how it is implemented. Instead, what we need to know is what operations objects of that type can perform.

Every class defines a type. The type name is the same as the name of the class. Hence, our `Sales_item` class defines a type named `Sales_item`. As with the built-in types, we can define a variable of a class type. When we write

```
Sales_item item;
```

we are saying that `item` is an object of type `Sales_item`. We often contract the phrase "an object of type `Sales_item`" to "a `Sales_item` object" or even more simply to "a `Sales_item`."

In addition to being able to define variables of type `Sales_item`, we can:

- Call a function named `isbn` to fetch the ISBN from a `Sales_item` object.
- Use the input (`>>`) and output (`<<`) operators to read and write objects of type `Sales_item`.
- Use the assignment operator (`=`) to assign one `Sales_item` object to another.
- Use the addition operator (`+`) to add two `Sales_item` objects. The two objects must refer to the same ISBN. The result is a new `Sales_item` object whose ISBN is that of its operands and whose number sold and revenue are the sum of the corresponding values in its operands.
- Use the compound assignment operator (`+=`) to add one `Sales_item` object into another.

### KEY CONCEPT: CLASSES DEFINE BEHAVIOR

The important thing to keep in mind when you read these programs is that the author of the `Sales_item` class defines *all* the actions that can be performed by objects of this class. That is, the `Sales_item` class defines what happens when a `Sales_item` object is created and what happens when the assignment, addition, or the input and output operators are applied to `Sales_items`.

In general, the class author determines all the operations that can be used on objects of the class type. For now, the only operations we know we can perform on `Sales_item` objects are the ones listed in this section.

## Reading and Writing `Sales_item`

Now that we know what operations we can use with `Sales_item` objects, we can write programs that use the class. For example, the following program reads data from the standard input into a `Sales_item` object and writes that `Sales_item` back onto the standard output:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // read ISBN, number of copies sold, and sales price
    std::cin >> book;
    // write ISBN, number of copies sold, total revenue, and average price
    std::cout << book << std::endl;
    return 0;
}
```

If the input to this program is

```
0-201-70353-X 4 24.99
```

then the output will be

```
0-201-70353-X 4 99.96 24.99
```

Our input says that we sold four copies of the book at \$24.99 each, and the output indicates that the total sold was four, the total revenue was \$99.96, and the average price per book was \$24.99.

This program starts with two `#include` directives, one of which uses a new form. Headers from the standard library are enclosed in angle brackets (`< >`). Those that are not part of the library are enclosed in double quotes (`" "`).

Inside `main` we define an object, named `book`, that we'll use to hold the data that we read from the standard input. The next statement reads into that object, and the third statement prints it to the standard output followed by printing `endl`.

## Adding `Sales_item`

A more interesting example adds two `Sales_item` objects:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;    // read a pair of transactions
    std::cout << item1 + item2 << std::endl; // print their sum
    return 0;
}
```

If we give this program the following input

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

our output is

```
0-201-78345-X 5 110 22
```

This program starts by including the `Sales_item` and `iostream` headers. Next we define two `Sales_item` objects to hold the transactions. We read data into these objects from the standard input. The output expression does the addition and prints the result.

It's worth noting how similar this program looks to the one on page 6: We read two inputs and write their sum. What makes this similarity noteworthy is that instead of reading and printing the sum of two integers, we're reading and printing the sum of two `Sales_item` objects. Moreover, the whole idea of "sum" is different. In the case of `ints` we are generating a conventional sum—the result of adding two numeric values. In the case of `Sales_item` objects we use a conceptually new meaning for sum—the result of adding the components of two `Sales_item` objects.

#### USING FILE REDIRECTION

It can be tedious to repeatedly type these transactions as input to the programs you are testing. Most operating systems support file redirection, which lets us associate a named file with the standard input and the standard output:

```
$ addItems <infile >outfile
```

Assuming `$` is the system prompt and our addition program has been compiled into an executable file named `addItems.exe` (or `addItems` on UNIX systems), this command will read transactions from a file named `infile` and write its output to a file named `outfile` in the current directory.

#### EXERCISES SECTION 1.5.1

**Exercise 1.20:** <http://www.informit.com/title/0321714113> contains a copy of `Sales_item.h` in the Chapter 1 code directory. Copy that file to your working directory. Use it to write a program that reads a set of book sales transactions, writing each transaction to the standard output.

**Exercise 1.21:** Write a program that reads two `Sales_item` objects that have the same ISBN and produces their sum.

**Exercise 1.22:** Write a program that reads several transactions for the same ISBN. Write the sum of all the transactions that were read.

## 1.5.2 A First Look at Member Functions

Our program that adds two `Sales_item`s should check whether the objects have the same ISBN. We'll do so as follows:

```
#include <iostream>
#include "Sales_item.h"

int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    // first check that item1 and item2 represent the same book
    if (item1.isbn() == item2.isbn()) {
        std::cout << item1 + item2 << std::endl;
        return 0;    // indicate success
    } else {
        std::cerr << "Data must refer to same ISBN"
                   << std::endl;
        return -1;   // indicate failure
    }
}
```

The difference between this program and the previous version is the `if` and its associated `else` branch. Even without understanding the `if` condition, we know what this program does. If the condition succeeds, then we write the same output as before and return 0, indicating success. If the condition fails, we execute the block following the `else`, which prints a message and returns an error indicator.

### What Is a Member Function?

The `if` condition

```
item1.isbn() == item2.isbn()
```

calls a **member function** named `isbn`. A member function is a function that is defined as part of a class. Member functions are sometimes referred to as **methods**.

Ordinarily, we call a member function on behalf of an object. For example, the first part of the left-hand operand of the equality expression

```
item1.isbn
```

uses the dot operator (the **“.” operator**) to say that we want “the `isbn` member of the object named `item1`.” The dot operator applies only to objects of class type. The left-hand operand must be an object of class type, and the right-hand operand must name a member of that type. The result of the dot operator is the member named by the right-hand operand.

When we use the dot operator to access a member function, we usually do so to call that function. We call a function using the call operator (the **() operator**). The call operator is a pair of parentheses that enclose a (possibly empty) list of *arguments*. The `isbn` member function does not take an argument. Thus,

```
item1.isbn()
```

calls the `isbn` function that is a member of the object named `item1`. This function returns the ISBN stored in `item1`.

The right-hand operand of the equality operator executes in the same way—it returns the ISBN stored in `item2`. If the ISBNs are the same, the condition is `true`; otherwise it is `false`.

## EXERCISES SECTION 1.5.2

**Exercise 1.23:** Write a program that reads several transactions and counts how many transactions occur for each ISBN.

**Exercise 1.24:** Test the previous program by giving multiple transactions representing multiple ISBNs. The records for each ISBN should be grouped together.

## 1.6 The Bookstore Program

We are now ready to solve our original bookstore problem. We need to read a file of sales transactions and produce a report that shows, for each book, the total number of copies sold, the total revenue, and the average sales price. We'll assume that all the transactions for each ISBN are grouped together in the input.

Our program will combine the data for each ISBN in a variable named `total`. We'll use a second variable named `trans` to hold each transaction we read. If `trans` and `total` refer to the same ISBN, we'll update `total`. Otherwise we'll print `total` and reset it using the transaction we just read:

```
#include <iostream>
#include "Sales_item.h"

int main()
{
    Sales_item total; // variable to hold data for the next transaction
    // read the first transaction and ensure that there are data to process
    if (std::cin >> total) {
        Sales_item trans; // variable to hold the running sum
        // read and process the remaining transactions
        while (std::cin >> trans) {
            // if we're still processing the same book
            if (total.isbn() == trans.isbn())
                total += trans; // update the running total
            else {
                // print results for the previous book
                std::cout << total << std::endl;
                total = trans; // total now refers to the next book
            }
        }
        std::cout << total << std::endl; // print the last transaction
    } else {
```

```
        // no input! warn the user
        std::cerr << "No data?!" << std::endl;
        return -1; // indicate failure
    }
    return 0;
}
```

This program is the most complicated one we've seen so far, but it uses only facilities that we have already seen.

As usual, we begin by including the headers that we use, `iostream` from the library and our own `Sales_item.h`. Inside `main` we define an object named `total`, which we'll use to sum the data for a given ISBN. We start by reading the first transaction into `total` and testing whether the read was successful. If the read fails, then there are no records and we fall through to the outermost `else` branch, which tells the user that there was no input.

Assuming we have successfully read a record, we execute the block following the outermost `if`. That block starts by defining the object named `trans`, which will hold our transactions as we read them. The `while` statement will read all the remaining records. As in our earlier programs, the `while` condition reads a value from the standard input. In this case, we read a `Sales_item` object into `trans`. As long as the read succeeds, we execute the body of the `while`.

The body of the `while` is a single `if` statement. The `if` checks whether the ISBNs are equal. If so, we use the compound assignment operator to add `trans` to `total`. If the ISBNs are not equal, we print the value stored in `total` and reset `total` by assigning `trans` to it. After executing the `if`, we return to the condition in the `while`, reading the next transaction, and so on until we run out of records.

When the `while` terminates, `total` contains the data for the last ISBN in the file. We write the data for the last ISBN in the last statement of the block that concludes the outermost `if` statement.

## EXERCISES SECTION 1.6

**Exercise 1.25:** Using the `Sales_item.h` header from the Web site, compile and execute the bookstore program presented in this section.

## CHAPTER SUMMARY

---

This chapter introduced enough of C++ to let you compile and execute simple C++ programs. We saw how to define a `main` function, which is the function that the operating system calls to execute our program. We also saw how to define variables, how to do input and output, and how to write `if`, `for`, and `while` statements. The chapter closed by introducing the most fundamental facility in C++: the class. In this chapter, we saw how to create and use objects of a class that someone else has defined. Later chapters will show how to define our own classes.

## DEFINED TERMS

---

**argument** Value passed to a function.

**assignment** Obliterates an object's current value and replaces that value by a new one.

**block** Sequence of zero or more statements enclosed in curly braces.

**buffer** A region of storage used to hold data. IO facilities often store input (or output) in a buffer and read or write the buffer independently from actions in the program. Output buffers can be explicitly flushed to force the buffer to be written. By default, reading `cin` flushes `cout`; `cout` is also flushed when the program ends normally.

**built-in type** Type, such as `int`, defined by the language.

**cerr** ostream object tied to the standard error, which often writes to the same device as the standard output. By default, writes to `cerr` are not buffered. Usually used for error messages or other output that is not part of the normal logic of the program.

**character string literal** Another term for string literal.

**cin** istream object used to read from the standard input.

**class** Facility for defining our own data structures together with associated operations. The class is one of the most fundamental features in C++. Library types, such as `istream` and `ostream`, are classes.

**class type** A type defined by a class. The name of the type is the class name.

**clog** ostream object tied to the standard error. By default, writes to `clog` are buffered. Usually used to report information about program execution to a log file.

**comments** Program text that is ignored by the compiler. C++ has two kinds of comments: single-line and paired. Single-line comments start with a `//`. Everything from the `//` to the end of the line is a comment. Paired comments begin with a `/*` and include all text up to the next `*/`.

**condition** An expression that is evaluated as true or false. A value of zero is false; any other value yields true.

**cout** ostream object used to write to the standard output. Ordinarily used to write the output of a program.

**curly brace** Curly braces delimit blocks. An open curly (`{`) starts a block; a close curly (`}`) ends one.

**data structure** A logical grouping of data and operations on that data.

**edit-compile-debug** The process of getting a program to execute properly.

**end-of-file** System-specific marker that indicates that there is no more input in a file.

**expression** The smallest unit of computation. An expression consists of one or more operands and usually one or more operators. Expressions are evaluated to produce a result. For example, assuming `i` and `j` are `ints`, then `i + j` is an expression and yields the sum of the two `int` values.

**for statement** Iteration statement that provides iterative execution. Often used to repeat a calculation a fixed number of times.

**function** Named unit of computation.

**function body** Block that defines the actions performed by a function.

**function name** Name by which a function is known and can be called.

**header** Mechanism whereby the definitions of a class or other names are made available to multiple programs. A program uses a header through a `#include` directive.

**if statement** Conditional execution based on the value of a specified condition. If the condition is true, the `if` body is executed. If not, the `else` body is executed if there is one.

**initialize** Give an object a value at the same time that it is created.

**iostream** Header that provides the library types for stream-oriented input and output.

**istream** Library type providing stream-oriented input.

**library type** Type, such as `istream`, defined by the standard library.

**main** Function called by the operating system to execute a C++ program. Each program must have one and only one function named `main`.

**manipulator** Object, such as `std::endl`, that when read or written “manipulates” the stream itself.

**member function** Operation defined by a class. Member functions ordinarily are called to operate on a specific object.

**method** Synonym for member function.

**namespace** Mechanism for putting names defined by a library into a single place. Namespaces help avoid inadvertent name clashes. The names defined by the C++ library are in the namespace `std`.

**ostream** Library type providing stream-oriented output.

**parameter list** Part of the definition of a function. Possibly empty list that specifies what arguments can be used to call the function.

**return type** Type of the value returned by a function.

**source file** Term used to describe a file that contains a C++ program.

**standard error** Output stream used for error reporting. Ordinarily, the standard output and the standard error are tied to the window in which the program is executed.

**standard input** Input stream usually associated with the window in which the program executes.

**standard library** Collection of types and functions that every C++ compiler must support. The library provides the types that support IO. C++ programmers tend to talk about “the library,” meaning the entire standard library. They also tend to refer to particular parts of the library by referring to a library type, such as the “`iostream` library,” meaning the part of the standard library that defines the IO classes.

**standard output** Output stream usually associated with the window in which the program executes.

**statement** A part of a program that specifies an action to take place when the program is executed. An expression followed by a semicolon is a statement; other kinds



of statements include blocks and `if`, `for`, and `while` statements, all of which contain other statements within themselves.

**std** Name of the namespace used by the standard library. `std::cout` indicates that we're using the name `cout` defined in the `std` namespace.

**string literal** Sequence of zero or more characters enclosed in double quotes ("a string literal").

**uninitialized variable** Variable that is not given an initial value. Variables of class type for which no initial value is specified are initialized as specified by the class definition. Variables of built-in type defined inside a function are uninitialized unless explicitly initialized. It is an error to try to use the value of an uninitialized variable. *Uninitialized variables are a rich source of bugs.*

**variable** A named object.

**while statement** Iteration statement that provides iterative execution so long as a specified condition is true. The body is executed zero or more times, depending on the truth value of the condition.

**() operator** Call operator. A pair of parentheses "()" following a function name. The operator causes a function to be invoked. Arguments to the function may be passed inside the parentheses.

**++ operator** Increment operator. Adds 1 to the operand; `++i` is equivalent to `i = i + 1`.

**+= operator** Compound assignment operator that adds the right-hand operand to the left and stores the result in the left-hand operand; `a += b` is equivalent to `a = a + b`.

**. operator** Dot operator. Left-hand operand must be an object of class type and the right-hand operand must be the name of a member of that object. The operator yields the named member of the given object.

**:: operator** Scope operator. Among other uses, the scope operator is used to access names in a namespace. For example,

`std::cout` denotes the name `cout` from the namespace `std`.

**= operator** Assigns the value of the right-hand operand to the object denoted by the left-hand operand.

**-- operator** Decrement operator. Subtracts 1 from the operand; `--i` is equivalent to `i = i - 1`.

**<< operator** Output operator. Writes the right-hand operand to the output stream indicated by the left-hand operand: `cout << "hi"` writes `hi` to the standard output. Output operations can be chained together: `cout << "hi" << "bye"` writes `hibye`.

**>> operator** Input operator. Reads from the input stream specified by the left-hand operand into the right-hand operand: `cin >> i` reads the next value on the standard input into `i`. Input operations can be chained together: `cin >> i >> j` reads first into `i` and then into `j`.

**#include** Directive that makes code in a header available to a program.

**== operator** The equality operator. Tests whether the left-hand operand is equal to the right-hand operand.

**!= operator** The inequality operator. Tests whether the left-hand operand is not equal to the right-hand operand.

**<= operator** The less-than-or-equal operator. Tests whether the left-hand operand is less than or equal to the right-hand operand.

**< operator** The less-than operator. Tests whether the left-hand operand is less than the right-hand operand.

**>= operator** Greater-than-or-equal operator. Tests whether the left-hand operand is greater than or equal to the right-hand operand.

**> operator** Greater-than operator. Tests whether the left-hand operand is greater than the right-hand operand.

# PART I

## THE BASICS

### CONTENTS

---

<b>Chapter 2</b>	<b>Variables and Basic Types . . . . .</b>	<b>31</b>
<b>Chapter 3</b>	<b>Strings, Vectors, and Arrays . . . . .</b>	<b>81</b>
<b>Chapter 4</b>	<b>Expressions . . . . .</b>	<b>133</b>
<b>Chapter 5</b>	<b>Statements . . . . .</b>	<b>171</b>
<b>Chapter 6</b>	<b>Functions . . . . .</b>	<b>201</b>
<b>Chapter 7</b>	<b>Classes . . . . .</b>	<b>253</b>

Every widely used programming language provides a common set of features, which differ in detail from one language to another. Understanding the details of how a language provides these features is the first step toward understanding the language. Among the most fundamental of these common features are

- Built-in types such as integers, characters, and so forth
- Variables, which let us give names to the objects we use
- Expressions and statements to manipulate values of these types
- Control structures, such as `if` or `while`, that allow us to conditionally or repeatedly execute a set of actions
- Functions that let us define callable units of computation

Most programming languages supplement these basic features in two ways: They let programmers extend the language by defining their own types, and they provide library routines that define useful functions and types not otherwise built into the language.

In C++, as in most programming languages, the type of an object determines what operations can be performed on it. Whether a particular expression is legal depends on the type of the objects in that expression. Some languages, such as Smalltalk and Python, check types at run time. In contrast, C++ is a statically typed language; type checking is done at compile time. As a consequence, the compiler must know the type of every name used in the program.

C++ provides a set of built-in types, operators to manipulate those types, and a small set of statements for program flow control. These elements form an alphabet from which we can write large, complicated, real-world systems. At this basic level, C++ is a simple language. Its expressive power arises from its support for mechanisms that allow the programmer to define new data structures. Using these facilities, programmers can shape the language to their own purposes without the language designers having to anticipate the programmers' needs.

Perhaps the most important feature in C++ is the class, which lets programmers define their own types. In C++ such types are sometimes called "class types" to distinguish them from the types that are built into the language. Some languages let programmers define types that specify only what data make up the type. Others, like C++, allow programmers to define types that include operations as well as data. A major design goal of C++ is to let programmers define their own types that are as easy to use as the built-in types. The Standard C++ library uses these features to implement a rich library of class types and associated functions.

The first step in mastering C++—learning the basics of the language and library—is the topic of Part I. Chapter 2 covers the built-in types and looks briefly at the mechanisms for defining our own new types. Chapter 3 introduces two of the most fundamental library types: `string` and `vector`. That chapter also covers arrays, which are a lower-level data structure built into C++ and many other languages. Chapters 4 through 6 cover expressions, statements, and functions. This part concludes in Chapter 7, which describes the basics of building our own class types. As we'll see, defining our own types brings together all that we've learned before, because writing a class entails using the facilities covered in Part I.

C H A P T E R

2

VARIABLES AND BASIC TYPES

CONTENTS

---

Section 2.1	Primitive Built-in Types . . . . .	32
Section 2.2	Variables . . . . .	41
Section 2.3	Compound Types . . . . .	50
Section 2.4	<code>const</code> Qualifier . . . . .	59
Section 2.5	Dealing with Types . . . . .	67
Section 2.6	Defining Our Own Data Structures . . . .	72
Chapter Summary	. . . . .	78
Defined Terms	. . . . .	78

Types are fundamental to any program: They tell us what our data mean and what operations we can perform on those data.

C++ has extensive support for types. The language defines several primitive types (characters, integers, floating-point numbers, etc.) and provides mechanisms that let us define our own data types. The library uses these mechanisms to define more complicated types such as variable-length character strings, vectors, and so on. This chapter covers the built-in types and begins our coverage of how C++ supports more complicated types.

*Types determine* the meaning of the data and operations in our programs. The meaning of even as simple a statement as

```
i = i + j;
```

depends on the types of `i` and `j`. If `i` and `j` are integers, this statement has the ordinary, arithmetic meaning of `+`. However, if `i` and `j` are `Sales_item` objects (§ 1.5.1, p. 20), this statement adds the components of these two objects.

## 2.1 Primitive Built-in Types

C++ defines a set of primitive types that include the **arithmetic types** and a special type named `void`. The arithmetic types represent characters, integers, boolean values, and floating-point numbers. The `void` type has no associated values and can be used in only a few circumstances, most commonly as the return type for functions that do not return a value.



### 2.1.1 Arithmetic Types

The arithmetic types are divided into two categories: **integral types** (which include character and boolean types) and floating-point types.

The size of—that is, the number of bits in—the arithmetic types varies across machines. The standard guarantees minimum sizes as listed in Table 2.1. However, compilers are allowed to use larger sizes for these types. Because the number of bits varies, the largest (or smallest) value that a type can represent also varies.

Table 2.1: C++: Arithmetic Types		
Type	Meaning	Minimum Size
<code>bool</code>	boolean	NA
<code>char</code>	character	8 bits
<code>wchar_t</code>	wide character	16 bits
<code>char16_t</code>	Unicode character	16 bits
<code>char32_t</code>	Unicode character	32 bits
<code>short</code>	short integer	16 bits
<code>int</code>	integer	16 bits
<code>long</code>	long integer	32 bits
<code>long long</code>	long integer	64 bits
<code>float</code>	single-precision floating-point	6 significant digits
<code>double</code>	double-precision floating-point	10 significant digits
<code>long double</code>	extended-precision floating-point	10 significant digits

The `bool` type represents the truth values `true` and `false`.

There are several character types, most of which exist to support internationalization. The basic character type is `char`. A `char` is guaranteed to be big enough to hold numeric values corresponding to the characters in the machine’s basic character set. That is, a `char` is the same size as a single machine byte.

The remaining character types—`wchar_t`, `char16_t`, and `char32_t`—are used for extended character sets. The `wchar_t` type is guaranteed to be large enough to hold any character in the machine’s largest extended character set. The types `char16_t` and `char32_t` are intended for Unicode characters. (Unicode is a standard for representing characters used in essentially any natural language.)

The remaining integral types represent integer values of (potentially) different sizes. The language guarantees that an `int` will be at least as large as `short`, a `long` at least as large as an `int`, and `long long` at least as large as `long`. The type `long long` was introduced by the new standard.



MACHINE-LEVEL REPRESENTATION OF THE BUILT-IN TYPES

Computers store data as a sequence of bits, each holding a 0 or 1, such as

```
00011011011100010110010000111011 ...
```

Most computers deal with memory as chunks of bits of sizes that are powers of 2. The smallest chunk of addressable memory is referred to as a “byte.” The basic unit of storage, usually a small number of bytes, is referred to as a “word.” In C++ a byte has at least as many bits as are needed to hold a character in the machine’s basic character set. On most machines a byte contains 8 bits and a word is either 32 or 64 bits, that is, 4 or 8 bytes.

Most computers associate a number (called an “address”) with each byte in memory. On a machine with 8-bit bytes and 32-bit words, we might view a word of memory as follows

736424	0	0	1	1	1	0	1	1
736425	0	0	0	1	1	0	1	1
736426	0	1	1	1	0	0	0	1
736427	0	1	1	0	0	1	0	0

Here, the byte’s address is on the left, with the 8 bits of the byte following the address.

We can use an address to refer to any of several variously sized collections of bits starting at that address. It is possible to speak of the word at address 736424 or the byte at address 736427. To give meaning to memory at a given address, we must know the type of the value stored there. The type determines how many bits are used and how to interpret those bits.

If the object at location 736424 has type `float` and if `float`s on this machine are stored in 32 bits, then we know that the object at that address spans the entire word. The value of that `float` depends on the details of how the machine stores floating-point numbers. Alternatively, if the object at location 736424 is an unsigned `char` on a machine using the ISO-Latin-1 character set, then the byte at that address represents a semicolon.

The floating-point types represent single-, double-, and extended-precision values. The standard specifies a minimum number of significant digits. Most compilers provide more precision than the specified minimum. Typically, `float`s are represented in one word (32 bits), `doubles` in two words (64 bits), and `long doubles` in either three or four words (96 or 128 bits). The `float` and `double` types typically yield about 7 and 16 significant digits, respectively. The type `long double`

is often used as a way to accommodate special-purpose floating-point hardware; its precision is more likely to vary from one implementation to another.

## Signed and Unsigned Types

Except for `bool` and the extended character types, the integral types may be **signed** or **unsigned**. A signed type represents negative or positive numbers (including zero); an unsigned type represents only values greater than or equal to zero.

The types `int`, `short`, `long`, and `long long` are all signed. We obtain the corresponding unsigned type by adding `unsigned` to the type, such as `unsigned long`. The type `unsigned int` may be abbreviated as `unsigned`.

Unlike the other integer types, there are three distinct basic character types: `char`, `signed char`, and `unsigned char`. In particular, `char` is not the same type as `signed char`. Although there are three character types, there are only two representations: signed and unsigned. The (plain) `char` type uses one of these representations. Which of the other two character representations is equivalent to `char` depends on the compiler.

In an unsigned type, all the bits represent the value. For example, an 8-bit `unsigned char` can hold the values from 0 through 255 inclusive.

The standard does not define how signed types are represented, but does specify that the range should be evenly divided between positive and negative values. Hence, an 8-bit `signed char` is guaranteed to be able to hold values from -127 through 127; most modern machines use representations that allow values from -128 through 127.

### ADVICE: DECIDING WHICH TYPE TO USE

C++, like C, is designed to let programs get close to the hardware when necessary. The arithmetic types are defined to cater to the peculiarities of various kinds of hardware. Accordingly, the number of arithmetic types in C++ can be bewildering. Most programmers can (and should) ignore these complexities by restricting the types they use. A few rules of thumb can be useful in deciding which type to use:

- Use an unsigned type when you know that the values cannot be negative.
- Use `int` for integer arithmetic. `short` is usually too small and, in practice, `long` often has the same size as `int`. If your data values are larger than the minimum guaranteed size of an `int`, then use `long long`.
- Do not use plain `char` or `bool` in arithmetic expressions. Use them *only* to hold characters or truth values. Computations using `char` are especially problematic because `char` is **signed** on some machines and **unsigned** on others. If you need a tiny integer, explicitly specify either `signed char` or `unsigned char`.
- Use `double` for floating-point computations; `float` usually does not have enough precision, and the cost of double-precision calculations versus single-precision is negligible. In fact, on some machines, double-precision operations are faster than single. The precision offered by `long double` usually is unnecessary and often entails considerable run-time cost.

**EXERCISES SECTION 2.1.1**

**Exercise 2.1:** What are the differences between `int`, `long`, `long long`, and `short`? Between an unsigned and a signed type? Between a `float` and a `double`?

**Exercise 2.2:** To calculate a mortgage payment, what types would you use for the rate, principal, and payment? Explain why you selected each type.

**2.1.2 Type Conversions**

The type of an object defines the data that an object might contain and what operations that object can perform. Among the operations that many types support is the ability to **convert** objects of the given type to other, related types.

Type conversions happen automatically when we use an object of one type where an object of another type is expected. We'll have more to say about conversions in § 4.11 (p. 159), but for now it is useful to understand what happens when we assign a value of one type to an object of another type.

When we assign one arithmetic type to another:

```
bool b = 42;           // b is true
int i = b;             // i has value 1
i = 3.14;              // i has value 3
double pi = i;         // pi has value 3.0
unsigned char c = -1;   // assuming 8-bit chars, c has value 255
signed char c2 = 256;   // assuming 8-bit chars, the value of c2 is undefined
```

what happens depends on the range of the values that the types permit:

- When we assign one of the nonbool arithmetic types to a bool object, the result is `false` if the value is 0 and `true` otherwise.
- When we assign a bool to one of the other arithmetic types, the resulting value is 1 if the bool is `true` and 0 if the bool is `false`.
- When we assign a floating-point value to an object of integral type, the value is truncated. The value that is stored is the part before the decimal point.
- When we assign an integral value to an object of floating-point type, the fractional part is zero. Precision may be lost if the integer has more bits than the floating-point object can accommodate.
- If we assign an out-of-range value to an object of unsigned type, the result is the remainder of the value modulo the number of values the target type can hold. For example, an 8-bit unsigned char can hold values from 0 through 255, inclusive. If we assign a value outside this range, the compiler assigns the remainder of that value modulo 256. Therefore, assigning -1 to an 8-bit unsigned char gives that object the value 255.
- If we assign an out-of-range value to an object of signed type, the result is **undefined**. The program might appear to work, it might crash, or it might produce garbage values.



**ADVICE: AVOID UNDEFINED AND IMPLEMENTATION-DEFINED BEHAVIOR**

Undefined behavior results from errors that the compiler is not required (and sometimes is not able) to detect. Even if the code compiles, a program that executes an undefined expression is in error.

Unfortunately, programs that contain undefined behavior can appear to execute correctly in some circumstances and/or on some compilers. There is no guarantee that the same program, compiled under a different compiler or even a subsequent release of the same compiler, will continue to run correctly. Nor is there any guarantee that what works with one set of inputs will work with another.

Similarly, programs usually should avoid implementation-defined behavior, such as assuming that the size of an `int` is a fixed and known value. Such programs are said to be *nonportable*. When the program is moved to another machine, code that relied on implementation-defined behavior may fail. Tracking down these sorts of problems in previously working programs is, mildly put, unpleasant.

The compiler applies these same type conversions when we use a value of one arithmetic type where a value of another arithmetic type is expected. For example, when we use a `nonbool` value as a condition (§ 1.4.1, p. 12), the arithmetic value is converted to `bool` in the same way that it would be converted if we had assigned that arithmetic value to a `bool` variable:

```
int i = 42;
if (i) // condition will evaluate as true
    i = 0;
```

If the value is 0, then the condition is `false`; all other (nonzero) values yield `true`.

By the same token, when we use a `bool` in an arithmetic expression, its value always converts to either 0 or 1. As a result, using a `bool` in an arithmetic expression is almost surely incorrect.



## Expressions Involving Unsigned Types

Although we are unlikely to intentionally assign a negative value to an object of unsigned type, we can (all too easily) write code that does so implicitly. For example, if we use both unsigned and `int` values in an arithmetic expression, the `int` value ordinarily is converted to unsigned. Converting an `int` to unsigned executes the same way as if we assigned the `int` to an unsigned:

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl; // prints -84
std::cout << u + i << std::endl; // if 32-bit ints, prints 4294967264
```

In the first expression, we add two (negative) `int` values and obtain the expected result. In the second expression, the `int` value `-42` is converted to unsigned before the addition is done. Converting a negative number to unsigned behaves exactly as if we had attempted to assign that negative value to an unsigned object. The value “wraps around” as described above.

Regardless of whether one or both operands are unsigned, if we subtract a value from an unsigned, we must be sure that the result cannot be negative:

```
unsigned u1 = 42, u2 = 10;
std::cout << u1 - u2 << std::endl; // ok: result is 32
std::cout << u2 - u1 << std::endl; // ok: but the result will wrap around
```

The fact that an unsigned cannot be less than zero also affects how we write loops. For example, in the exercises to § 1.4.1 (p. 13), you were to write a loop that used the decrement operator to print the numbers from 10 down to 0. The loop you wrote probably looked something like

```
for (int i = 10; i >= 0; --i)
    std::cout << i << std::endl;
```

We might think we could rewrite this loop using an unsigned. After all, we don't plan to print negative numbers. However, this simple change in type means that our loop will never terminate:

```
// WRONG: u can never be less than 0; the condition will always succeed
for (unsigned u = 10; u >= 0; --u)
    std::cout << u << std::endl;
```

Consider what happens when *u* is 0. On that iteration, we'll print 0 and then execute the expression in the *for* loop. That expression, *--u*, subtracts 1 from *u*. That result, -1, won't fit in an unsigned value. As with any other out-of-range value, -1 will be transformed to an unsigned value. Assuming 32-bit ints, the result of *--u*, when *u* is 0, is 4294967295.

One way to write this loop is to use a *while* instead of a *for*. Using a *while* lets us decrement before (rather than after) printing our value:

```
unsigned u = 11; // start the loop one past the first element we want to print
while (u > 0) {
    --u;          // decrement first, so that the last iteration will print 0
    std::cout << u << std::endl;
}
```

This loop starts by decrementing the value of the loop control variable. On the last iteration, *u* will be 1 on entry to the loop. We'll decrement that value, meaning that we'll print 0 on this iteration. When we next test *u* in the *while* condition, its value will be 0 and the loop will exit. Because we start by decrementing *u*, we have to initialize *u* to a value one greater than the first value we want to print. Hence, we initialize *u* to 11, so that the first value printed is 10.

### CAUTION: DON'T MIX SIGNED AND UNSIGNED TYPES

Expressions that mix signed and unsigned values can yield surprising results when the signed value is negative. It is essential to remember that signed values are automatically converted to unsigned. For example, in an expression like *a \* b*, if *a* is -1 and *b* is 1, then if both *a* and *b* are ints, the value is, as expected -1. However, if *a* is int and *b* is an unsigned, then the value of this expression depends on how many bits an int has on the particular machine. On our machine, this expression yields 4294967295.

## EXERCISES SECTION 2.1.2

**Exercise 2.3:** What output will the following code produce?

```
unsigned u = 10, u2 = 42;
std::cout << u2 - u << std::endl;
std::cout << u - u2 << std::endl;

int i = 10, i2 = 42;
std::cout << i2 - i << std::endl;
std::cout << i - i2 << std::endl;

std::cout << i - u << std::endl;
std::cout << u - i << std::endl;
```

**Exercise 2.4:** Write a program to check whether your predictions were correct. If not, study this section until you understand what the problem is.

### 2.1.3 Literals

A value, such as 42, is known as a **literal** because its value self-evident. Every literal has a type. The form and value of a literal determine its type.

#### Integer and Floating-Point Literals

We can write an integer literal using decimal, octal, or hexadecimal notation. Integer literals that begin with 0 (zero) are interpreted as octal. Those that begin with either 0x or 0X are interpreted as hexadecimal. For example, we can write the value 20 in any of the following three ways:

```
20 /* decimal */ 024 /* octal */ 0x14 /* hexadecimal */
```

The type of an integer literal depends on its value and notation. By default, decimal literals are signed whereas octal and hexadecimal literals can be either signed or unsigned types. A decimal literal has the smallest type of `int`, `long`, or `long long` (i.e., the first type in this list) in which the literal's value fits. Octal and hexadecimal literals have the smallest type of `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` in which the literal's value fits. It is an error to use a literal that is too large to fit in the largest related type. There are no literals of type `short`. We'll see in Table 2.2 (p. 40) that we can override these defaults by using a suffix.

Although integer literals may be stored in signed types, technically speaking, the value of a decimal literal is never a negative number. If we write what appears to be a negative decimal literal, for example, `-42`, the minus sign is *not* part of the literal. The minus sign is an operator that negates the value of its (literal) operand.

Floating-point literals include either a decimal point or an exponent specified using scientific notation. Using scientific notation, the exponent is indicated by either `E` or `e`:

```
3.14159      3.14159E0      0.      0e0      .001
```

By default, floating-point literals have type `double`. We can override the default using a suffix from Table 2.2 (overleaf).

## Character and Character String Literals

A character enclosed within single quotes is a literal of type `char`. Zero or more characters enclosed in double quotation marks is a string literal:

```
'a'    // character literal
"Hello World!" // string literal
```

The type of a string literal is *array* of constant chars, a type we'll discuss in § 3.5.4 (p. 122). The compiler appends a null character (`'\0'`) to every string literal. Thus, the actual size of a string literal is one more than its apparent size. For example, the literal `'A'` represents the single character A, whereas the string literal `"A"` represents an array of two characters, the letter A and the null character.

Two string literals that appear adjacent to one another and that are separated only by spaces, tabs, or newlines are concatenated into a single literal. We use this form of literal when we need to write a literal that would otherwise be too large to fit comfortably on a single line:

```
// multiline string literal
std::cout << "a really, really long string literal "
              "that spans two lines" << std::endl;
```

## Escape Sequences

Some characters, such as backspace or control characters, have no visible image. Such characters are **nonprintable**. Other characters (single and double quotation marks, question mark, and backslash) have special meaning in the language. Our programs cannot use any of these characters directly. Instead, we use an **escape sequence** to represent such characters. An escape sequence begins with a backslash. The language defines several escape sequences:

newline	<code>\n</code>	horizontal tab	<code>\t</code>	alert (bell)	<code>\a</code>
vertical tab	<code>\v</code>	backspace	<code>\b</code>	double quote	<code>\"</code>
backslash	<code>\\</code>	question mark	<code>\?</code>	single quote	<code>\'</code>
carriage return	<code>\r</code>	formfeed	<code>\f</code>		

We use an escape sequence as if it were a single character:

```
std::cout << '\n';           // prints a newline
std::cout << "\tHi!\n";      // prints a tab followed by "Hi!" and a newline
```

We can also write a generalized escape sequence, which is `\x` followed by one or more hexadecimal digits or a `\` followed by one, two, or three octal digits. The value represents the numerical value of the character. Some examples (assuming the Latin-1 character set):

<code>\7</code> (bell)	<code>\12</code> (newline)	<code>\40</code> (blank)
<code>\0</code> (null)	<code>\115</code> ('M')	<code>\x4d</code> ('M')

As with an escape sequence defined by the language, we use these escape sequences as we would any other character:

```
std::cout << "Hi \x4d0\115!\n"; // prints Hi MOM! followed by a newline
std::cout << '\115' << '\n';    // prints M followed by a newline
```

Note that if a `\` is followed by more than three octal digits, only the first three are associated with the `\`. For example, `"\1234"` represents two characters: the character represented by the octal value 123 and the character 4. In contrast, `\x` uses up all the hex digits following it; `"\x1234"` represents a single, 16-bit character composed from the bits corresponding to these four hexadecimal digits. Because most machines have 8-bit chars, such values are unlikely to be useful. Ordinarily, hexadecimal characters with more than 8 bits are used with extended characters sets using one of the prefixes from Table 2.2.

Specifying the Type of a Literal

We can override the default type of an integer, floating-point, or character literal by supplying a suffix or prefix as listed in Table 2.2.

```
L'a'           // wide character literal, type is wchar_t
u8"hi!"        // utf-8 string literal (utf-8 encodes a Unicode character in 8 bits)
42ULL          // unsigned integer literal, type is unsigned long long
1E-3F          // single-precision floating-point literal, type is float
3.14159L       // extended-precision floating-point literal, type is long double
```


When you write a long literal, use the uppercase `L`; the lowercase letter `l` is too easily mistaken for the digit 1.

Table 2.2: Specifying the Type of a Literal				
Character and Character String Literals				
Prefix	Meaning	Type		
u	Unicode 16 character	char16_t		
U	Unicode 32 character	char32_t		
L	wide character	wchar_t		
u8	utf-8 (string literals only)	char		
Integer Literals		Floating-Point Literals		
Suffix	Minimum Type	Suffix	Type	
u or U	unsigned	f or F	float	
l or L	long	l or L	long double	
ll or LL	long long			

We can independently specify the signedness and size of an integral literal. If the suffix contains a `U`, then the literal has an unsigned type, so a decimal, octal, or hexadecimal literal with a `U` suffix has the smallest type of unsigned `int`, unsigned `long`, or unsigned `long long` in which the literal's value fits. If the suffix contains an `L`, then the literal's type will be at least `long`; if the suffix contains `LL`, then the literal's type will be either `long long` or unsigned `long long`.

We can combine U with either L or LL. For example, a literal with a suffix of UL will be either unsigned long or unsigned long long, depending on whether its value fits in unsigned long.

## Boolean and Pointer Literals

The words `true` and `false` are literals of type `bool`:

```
bool test = false;
```

The word `nullptr` is a pointer literal. We'll have more to say about pointers and `nullptr` in § 2.3.2 (p. 52).

### EXERCISES SECTION 2.1.3

**Exercise 2.5:** Determine the type of each of the following literals. Explain the differences among the literals in each of the four examples:

- (a) `'a'`, `L'a'`, `"a"`, `L"a"`
- (b) `10`, `10u`, `10L`, `10uL`, `012`, `0xC`
- (c) `3.14`, `3.14f`, `3.14L`
- (d) `10`, `10u`, `10.`, `10e-2`

**Exercise 2.6:** What, if any, are the differences between the following definitions:

```
int month = 9, day = 7;
int month = 09, day = 07;
```

**Exercise 2.7:** What values do these literals represent? What type does each have?

- (a) `"Who goes with F\145rgus?\012"`
- (b) `3.14e1L`      (c) `1024f`      (d) `3.14L`

**Exercise 2.8:** Using escape sequences, write a program to print 2M followed by a newline. Modify the program to print 2, then a tab, then an M, followed by a newline.

## 2.2 Variables

A *variable* provides us with named storage that our programs can manipulate. Each variable in C++ has a type. The type determines the size and layout of the variable's memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable. C++ programmers tend to refer to variables as "variables" or "objects" interchangeably.

### 2.2.1 Variable Definitions

A simple variable definition consists of a **type specifier**, followed by a list of one or more variable names separated by commas, and ends with a semicolon. Each name



in the list has the type defined by the type specifier. A definition may (optionally) provide an initial value for one or more of the names it defines:

```
int sum = 0, value, // sum, value, and units_sold have type int
    units_sold = 0; // sum and units_sold have initial value 0
Sales_item item;    // item has type Sales_item (see § 1.5.1 (p. 20))
// string is a library type, representing a variable-length sequence of characters
std::string book("0-201-78345-X"); // book initialized from string literal
```

The definition of `book` uses the `std::string` library type. Like `iostream` (§ 1.2, p. 7), `string` is defined in namespace `std`. We'll have more to say about the `string` type in Chapter 3. For now, what's useful to know is that a `string` is a type that represents a variable-length sequence of characters. The `string` library gives us several ways to initialize `string` objects. One of these ways is as a copy of a `string` literal (§ 2.1.3, p. 39). Thus, `book` is initialized to hold the characters `0-201-78345-X`.

### TERMINOLOGY: WHAT IS AN OBJECT?

C++ programmers tend to be cavalier in their use of the term *object*. Most generally, an object is a region of memory that can contain data and has a type.

Some use the term *object* only to refer to variables or values of class types. Others distinguish between named and unnamed objects, using the term *variable* to refer to named objects. Still others distinguish between objects and values, using the term *object* for data that can be changed by the program and the term *value* for data that are read-only.

In this book, we'll follow the more general usage that an object is a region of memory that has a type. We will freely use the term *object* regardless of whether the object has built-in or class type, is named or unnamed, or can be read or written.

## Initializers

An object that is **initialized** gets the specified value at the moment it is created. The values used to initialize a variable can be arbitrarily complicated expressions. When a definition defines two or more variables, the name of each object becomes visible immediately. Thus, it is possible to initialize a variable to the value of one defined earlier in the same definition.

```
// ok: price is defined and initialized before it is used to initialize discount
double price = 109.99, discount = price * 0.16;
// ok: call applyDiscount and use the return value to initialize salePrice
double salePrice = applyDiscount(price, discount);
```

Initialization in C++ is a surprisingly complicated topic and one we will return to again and again. Many programmers are confused by the use of the `=` symbol to initialize a variable. It is tempting to think of initialization as a form of assignment, but initialization and assignment are different operations in C++. This concept is particularly confusing because in many languages the distinction is irrelevant

and can be ignored. Moreover, even in C++ the distinction often doesn't matter. Nonetheless, it is a crucial concept and one we will reiterate throughout the text.



Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replaces that value with a new one.

## List Initialization

One way in which initialization is a complicated topic is that the language defines several different forms of initialization. For example, we can use any of the following four different ways to define an `int` variable named `units_sold` and initialize it to 0:

```
int units_sold = 0;
int units_sold = {0};
int units_sold{0};
int units_sold(0);
```

The generalized use of curly braces for initialization was introduced as part of the new standard. This form of initialization previously had been allowed only in more restricted ways. For reasons we'll learn about in § 3.3.1 (p. 98), this form of initialization is referred to as **list initialization**. Braced lists of initializers can now be used whenever we initialize an object and in some cases when we assign a new value to an object.

C++  
11

When used with variables of built-in type, this form of initialization has one important property: The compiler will not let us list initialize variables of built-in type if the initializer might lead to the loss of information:

```
long double ld = 3.1415926536;
int a{ld}, b = {ld}; // error: narrowing conversion required
int c(ld), d = ld;   // ok: but value will be truncated
```

The compiler rejects the initializations of `a` and `b` because using a `long double` to initialize an `int` is likely to lose data. At a minimum, the fractional part of `ld` will be truncated. In addition, the integer part in `ld` might be too large to fit in an `int`.

As presented here, the distinction might seem trivial—after all, we'd be unlikely to directly initialize an `int` from a `long double`. However, as we'll see in Chapter 16, such initializations might happen unintentionally. We'll say more about these forms of initialization in § 3.2.1 (p. 84) and § 3.3.1 (p. 98).

## Default Initialization

When we define a variable without an initializer, the variable is **default initialized**. Such variables are given the "default" value. What that default value is depends on the type of the variable and may also depend on where the variable is defined.

The value of an object of built-in type that is not explicitly initialized depends on where it is defined. Variables defined outside any function body are initialized to zero. With one exception, which we cover in § 6.1.1 (p. 205), variables of built-in



type defined inside a function are **uninitialized**. The value of an uninitialized variable of built-in type is undefined (§ 2.1.2, p. 36). It is an error to copy or otherwise try to access the value of a variable whose value is undefined.

Each class controls how we initialize objects of that class type. In particular, it is up to the class whether we can define objects of that type without an initializer. If we can, the class determines what value the resulting object will have.

Most classes let us define objects without explicit initializers. Such classes supply an appropriate default value for us. For example, as we've just seen, the library `string` class says that if we do not supply an initializer, then the resulting `string` is the empty string:

```
std::string empty; // empty implicitly initialized to the empty string
Sales_item item;   // default-initialized Sales_item object
```

Some classes require that every object be explicitly initialized. The compiler will complain if we try to create an object of such a class with no initializer.



Uninitialized objects of built-in type defined inside a function body have undefined value. Objects of class type that we do not explicitly initialize have a value that is defined by the class.

## EXERCISES SECTION 2.2.1

**Exercise 2.9:** Explain the following definitions. For those that are illegal, explain what's wrong and how to correct it.

- (a) `std::cin >> int input_value;`      (b) `int i = { 3.14 };`  
 (c) `double salary = wage = 9999.99;`    (d) `int i = 3.14;`

**Exercise 2.10:** What are the initial values, if any, of each of the following variables?

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
}
```



## 2.2.2 Variable Declarations and Definitions

To allow programs to be written in logical parts, C++ supports what is commonly known as *separate compilation*. Separate compilation lets us split our programs into several files, each of which can be compiled independently.

When we separate a program into multiple files, we need a way to share code across those files. For example, code defined in one file may need to use a variable defined in another file. As a concrete example, consider `std::cout` and

**CAUTION: UNINITIALIZED VARIABLES CAUSE RUN-TIME PROBLEMS**

An uninitialized variable has an indeterminate value. Trying to use the value of an uninitialized variable is an error that is often hard to debug. Moreover, the compiler is not required to detect such errors, although most will warn about at least some uses of uninitialized variables.

What happens when we use an uninitialized variable is undefined. Sometimes, we're lucky and our program crashes as soon as we access the object. Once we track down the location of the crash, it is usually easy to see that the variable was not properly initialized. Other times, the program completes but produces erroneous results. Even worse, the results may appear correct on one run of our program but fail on a subsequent run. Moreover, adding code to the program in an unrelated location can cause what we thought was a correct program to start producing incorrect results.



We recommend initializing every object of built-in type. It is not always necessary, but it is easier and safer to provide an initializer until you can be certain it is safe to omit the initializer.

`std::cin`. These are objects defined somewhere in the standard library, yet our programs can use these objects.

To support separate compilation, C++ distinguishes between declarations and definitions. A **declaration** makes a name known to the program. A file that wants to use a name defined elsewhere includes a declaration for that name. A **definition** creates the associated entity.

A variable declaration specifies the type and name of a variable. A variable definition is a declaration. In addition to specifying the name and type, a definition also allocates storage and may provide the variable with an initial value.

To obtain a declaration that is not also a definition, we add the `extern` keyword and may not provide an explicit initializer:

```
extern int i;    // declares but does not define i
int j;          // declares and defines j
```

Any declaration that includes an explicit initializer is a definition. We can provide an initializer on a variable defined as `extern`, but doing so overrides the `extern`. An `extern` that has an initializer is a definition:

```
extern double pi = 3.1416; // definition
```

It is an error to provide an initializer on an `extern` inside a function.



Variables must be defined exactly once but can be declared many times.

The distinction between a declaration and a definition may seem obscure at this point but is actually important. To use a variable in more than one file requires declarations that are separate from the variable's definition. To use the same variable in multiple files, we must define that variable in one—and only one—file. Other files that use that variable must declare—but not define—that variable.

We'll have more to say about how C++ supports separate compilation in § 2.6.3 (p. 76) and § 6.1.3 (p. 207).

### EXERCISES SECTION 2.2.2

**Exercise 2.11:** Explain whether each of the following is a declaration or a definition:

- (a) `extern int ix = 1024;`
- (b) `int iy;`
- (c) `extern int iz;`

### KEY CONCEPT: STATIC TYPING

C++ is a *statically typed* language, which means that types are checked at compile time. The process by which types are checked is referred to as *type checking*.

As we've seen, the type of an object constrains the operations that the object can perform. In C++, the compiler checks whether the operations we write are supported by the types we use. If we try to do things that the type does not support, the compiler generates an error message and does not produce an executable file.

As our programs get more complicated, we'll see that static type checking can help find bugs. However, a consequence of static checking is that the type of every entity we use must be known to the compiler. As one example, we must declare the type of a variable before we can use that variable.

## 2.2.3 Identifiers

*Identifiers* in C++ can be composed of letters, digits, and the underscore character. The language imposes no limit on name length. Identifiers must begin with either a letter or an underscore. Identifiers are case-sensitive; upper- and lowercase letters are distinct:

```
// defines four different int variables
int somename, someName, SomeName, SOMENAME;
```


The language reserves a set of names, listed in Tables 2.3 and Table 2.4, for its own use. These names may not be used as identifiers.

The standard also reserves a set of names for use in the standard library. The identifiers we define in our own programs may not contain two consecutive underscores, nor can an identifier begin with an underscore followed immediately by an uppercase letter. In addition, identifiers defined outside a function may not begin with an underscore.

## Conventions for Variable Names

There are a number of generally accepted conventions for naming variables. Following these conventions can improve the readability of a program.

- An identifier should give some indication of its meaning.
- Variable names normally are lowercase—index, not Index or INDEX.
- Like Sales\_item, classes we define usually begin with an uppercase letter.
- Identifiers with multiple words should visually distinguish each word, for example, student\_loan or studentLoan, not studentloan.

Best Practices

Naming conventions are most useful when followed consistently.

Table 2.3: C++ Keywords				
alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

Table 2.4: C++ Alternative Operator Names					
and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

EXERCISES SECTION 2.2.3

**Exercise 2.12:** Which, if any, of the following names are invalid?

(a) int double = 3.14;

(b) int \_;

(c) int catch-22;

(d) int 1\_or\_2 = 1;

(e) double Double = 3.14;

## 2.2.4 Scope of a Name



At any particular point in a program, each name that is in use refers to a specific entity—a variable, function, type, and so on. However, a given name can be reused to refer to different entities at different points in the program.

A **scope** is a part of the program in which a name has a particular meaning. Most scopes in C++ are delimited by curly braces.

The same name can refer to different entities in different scopes. Names are visible from the point where they are declared until the end of the scope in which the declaration appears.

As an example, consider the program from § 1.4.2 (p. 13):

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

This program defines three names—`main`, `sum`, and `val`—and uses the namespace name `std`, along with two names from that namespace—`cout` and `endl`.

The name `main` is defined outside any curly braces. The name `main`—like most names defined outside a function—has **global scope**. Once declared, names at the global scope are accessible throughout the program. The name `sum` is defined within the scope of the block that is the body of the `main` function. It is accessible from its point of declaration throughout the rest of the `main` function but not outside of it. The variable `sum` has **block scope**. The name `val` is defined in the scope of the `for` statement. It can be used in that statement but not elsewhere in `main`.

### ADVICE: DEFINE VARIABLES WHERE YOU FIRST USE THEM

It is usually a good idea to define an object near the point at which the object is first used. Doing so improves readability by making it easy to find the definition of the variable. More importantly, it is often easier to give the variable a useful initial value when the variable is defined close to where it is first used.

## Nested Scopes

Scopes can contain other scopes. The contained (or nested) scope is referred to as an **inner scope**, the containing scope is the **outer scope**.

Once a name has been declared in a scope, that name can be used by scopes nested inside that scope. Names declared in the outer scope can also be redefined in an inner scope:

```

#include <iostream>
// Program for illustration purposes only: It is bad style for a function
// to use a global variable and also define a local variable with the same name
int reused = 42; // reused has global scope
int main()
{
    int unique = 0; // unique has block scope
    // output #1: uses global reused; prints 42 0
    std::cout << reused << " " << unique << std::endl;
    int reused = 0; // new, local object named reused hides global reused
    // output #2: uses local reused; prints 0 0
    std::cout << reused << " " << unique << std::endl;
    // output #3: explicitly requests the global reused; prints 42 0
    std::cout << ::reused << " " << unique << std::endl;
    return 0;
}

```

Output #1 appears before the local definition of `reused`. Therefore, this output statement uses the name `reused` that is defined in the global scope. This statement prints 42 0. Output #2 occurs after the local definition of `reused`. The local `reused` is now **in scope**. Thus, this second output statement uses the local object named `reused` rather than the global one and prints 0 0. Output #3 uses the scope operator (§ 1.2, p. 8) to override the default scoping rules. The global scope has no name. Hence, when the scope operator has an empty left-hand side, it is a request to fetch the name on the right-hand side from the global scope. Thus, this expression uses the global `reused` and prints 42 0.



WARNING

It is almost always a bad idea to define a local variable with the same name as a global variable that the function uses or might use.

## EXERCISES SECTION 2.2.4

**Exercise 2.13:** What is the value of `j` in the following program?

```

int i = 42;
int main()
{
    int i = 100;
    int j = i;
}

```

**Exercise 2.14:** Is the following program legal? If so, what values are printed?

```

int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << i << " " << sum << std::endl;

```



## 2.3 Compound Types

A **compound type** is a type that is defined in terms of another type. C++ has several compound types, two of which—references and pointers—we'll cover in this chapter.

Defining variables of compound type is more complicated than the declarations we've seen so far. In § 2.2 (p. 41) we said that simple declarations consist of a type followed by a list of variable names. More generally, a declaration is a **base type** followed by a list of **declarators**. Each declarator names a variable and gives the variable a type that is related to the base type.

The declarations we have seen so far have declarators that are nothing more than variable names. The type of such variables is the base type of the declaration. More complicated declarators specify variables with compound types that are built from the base type of the declaration.



### 2.3.1 References



The new standard introduced a new kind of reference: an “rvalue reference,” which we'll cover in § 13.6.1 (p. 532). These references are primarily intended for use inside classes. Technically speaking, when we use the term *reference*, we mean “lvalue reference.”

A **reference** defines an alternative name for an object. A reference type “refers to” another type. We define a reference type by writing a declarator of the form `&d`, where `d` is the name being declared:

```
int ival = 1024;
int &refVal = ival;    // refVal refers to (is another name for) ival
int &refVal2;          // error: a reference must be initialized
```

Ordinarily, when we initialize a variable, the value of the initializer is copied into the object we are creating. When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer. Once initialized, a reference remains bound to its initial object. There is no way to rebind a reference to refer to a different object. Because there is no way to rebind a reference, references *must* be initialized.



### A Reference Is an Alias



A reference is not an object. Instead, a reference is *just another name for an already existing object*.

After a reference has been defined, *all* operations on that reference are actually operations on the object to which the reference is bound:

```
refVal = 2;           // assigns 2 to the object to which refVal refers, i.e., to ival
int ii = refVal;      // same as ii = ival
```

When we assign to a reference, we are assigning to the object to which the reference is bound. When we fetch the value of a reference, we are really fetching the value of the object to which the reference is bound. Similarly, when we use a reference as an initializer, we are really using the object to which the reference is bound:

```
// ok: refVal3 is bound to the object to which refVal is bound, i.e., to ival
int &refVal3 = refVal;

// initializes i from the value in the object to which refVal is bound
int i = refVal; // ok: initializes i to the same value as ival
```

Because references are not objects, we may not define a reference to a reference.

## Reference Definitions

We can define multiple references in a single definition. Each identifier that is a reference must be preceded by the `&` symbol:

```
int i = 1024, i2 = 2048; // i and i2 are both ints
int &r = i, r2 = i2;     // r is a reference bound to i; r2 is an int
int i3 = 1024, &ri = i3; // i3 is an int; ri is a reference bound to i3
int &r3 = i3, &r4 = i2;  // both r3 and r4 are references
```

With two exceptions that we'll cover in § 2.4.1 (p. 61) and § 15.2.3 (p. 601), the type of a reference and the object to which the reference refers must match exactly. Moreover, for reasons we'll explore in § 2.4.1, a reference may be bound only to an object, not to a literal or to the result of a more general expression:

```
int &refVal4 = 10; // error: initializer must be an object
double dval = 3.14;
int &refVal5 = dval; // error: initializer must be an int object
```

### EXERCISES SECTION 2.3.1

**Exercise 2.15:** Which of the following definitions, if any, are invalid? Why?

- (a) `int ival = 1.01;`      (b) `int &rval1 = 1.01;`  
 (c) `int &rval2 = ival;`    (d) `int &rval3;`

**Exercise 2.16:** Which, if any, of the following assignments are invalid? If they are valid, explain what they do.

- ```
int i = 0, &r1 = i; double d = 0, &r2 = d;
```
- (a) `r2 = 3.14159;`      (b) `r2 = r1;`  
 (c) `i = r2;`            (d) `r1 = d;`

**Exercise 2.17:** What does the following code print?

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```





## 2.3.2 Pointers

A **pointer** is a compound type that “points to” another type. Like references, pointers are used for indirect access to other objects. Unlike a reference, a pointer is an object in its own right. Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime. Unlike a reference, a pointer need not be initialized at the time it is defined. Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.



Pointers are often hard to understand. Debugging problems due to pointer errors bedevil even experienced programmers.

We define a pointer type by writing a declarator of the form `*d`, where `d` is the name being defined. The `*` must be repeated for each pointer variable:

```
int *ip1, *ip2; // both ip1 and ip2 are pointers to int
double dp, *dp2; // dp2 is a pointer to double; dp is a double
```

### Taking the Address of an Object

A pointer holds the address of another object. We get the address of an object by using the address-of operator (the **& operator**):

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
```

The second statement defines `p` as a pointer to `int` and initializes `p` to point to the `int` object named `ival`. Because references are not objects, they don’t have addresses. Hence, we may not define a pointer to a reference.

With two exceptions, which we cover in § 2.4.2 (p. 62) and § 15.2.3 (p. 601), the types of the pointer and the object to which it points must match:

```
double dval;
double *pd = &dval; // ok: initializer is the address of a double
double *pd2 = pd;   // ok: initializer is a pointer to double
int *pi = pd;       // error: types of pi and pd differ
pi = &dval;          // error: assigning the address of a double to a pointer to int
```

The types must match because the type of the pointer is used to infer the type of the object to which the pointer points. If a pointer addressed an object of another type, operations performed on the underlying object would fail.

### Pointer Value

The value (i.e., the address) stored in a pointer can be in one of four states:

1. It can point to an object.
2. It can point to the location just immediately past the end of an object.
3. It can be a null pointer, indicating that it is not bound to any object.
4. It can be invalid; values other than the preceding three are invalid.

It is an error to copy or otherwise try to access the value of an invalid pointer. As when we use an uninitialized variable, this error is one that the compiler is unlikely to detect. The result of accessing an invalid pointer is undefined. Therefore, we must always know whether a given pointer is valid.

Although pointers in cases 2 and 3 are valid, there are limits on what we can do with such pointers. Because these pointers do not point to any object, we may not use them to access the (supposed) object to which the pointer points. If we do attempt to access an object through such pointers, the behavior is undefined.

## Using a Pointer to Access an Object

When a pointer points to an object, we can use the dereference operator (the **\*** operator) to access that object:

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
cout << *p;     // * yields the object to which p points; prints 42
```

Dereferencing a pointer yields the object to which the pointer points. We can assign to that object by assigning to the result of the dereference:

```
*p = 0;          // * yields the object; we assign a new value to ival through p
cout << *p;     // prints 0
```

When we assign to `*p`, we are assigning to the object to which `p` points.



We may dereference only a valid pointer that points to an object.

### KEY CONCEPT: SOME SYMBOLS HAVE MULTIPLE MEANINGS

Some symbols, such as `&` and `*`, are used as both an operator in an expression and as part of a declaration. The context in which a symbol is used determines what the symbol means:

```
int i = 42;
int &r = i; // & follows a type and is part of a declaration; r is a reference
int *p;    // * follows a type and is part of a declaration; p is a pointer
p = &i;    // & is used in an expression as the address-of operator
*p = i;    // * is used in an expression as the dereference operator
int &r2 = *p; // & is part of the declaration; * is the dereference operator
```

In declarations, `&` and `*` are used to form compound types. In expressions, these same symbols are used to denote an operator. Because the same symbol is used with very different meanings, it can be helpful to ignore appearances and think of them as if they were different symbols.

## Null Pointers

A **null pointer** does not point to any object. Code can check whether a pointer is null before attempting to use it. There are several ways to obtain a null pointer:

```
int *p1 = nullptr; // equivalent to int *p1 = 0;
int *p2 = 0;       // directly initializes p2 from the literal constant 0
// must #include cstdlib
int *p3 = NULL;    // equivalent to int *p3 = 0;
```

**C++  
11**

The most direct approach is to initialize the pointer using the literal `nullptr`, which was introduced by the new standard. `nullptr` is a literal that has a special type that can be converted (§ 2.1.2, p. 35) to any other pointer type. Alternatively, we can initialize a pointer to the literal `0`, as we do in the definition of `p2`.

Older programs sometimes use a **preprocessor variable** named `NULL`, which the `cstdlib` header defines as `0`.

We'll describe the preprocessor in a bit more detail in § 2.6.3 (p. 77). What's useful to know now is that the preprocessor is a program that runs before the compiler. Preprocessor variables are managed by the preprocessor, and are not part of the `std` namespace. As a result, we refer to them directly without the `std::` prefix.

When we use a preprocessor variable, the preprocessor automatically replaces the variable by its value. Hence, initializing a pointer to `NULL` is equivalent to initializing it to `0`. Modern C++ programs generally should avoid using `NULL` and use `nullptr` instead.

It is illegal to assign an `int` variable to a pointer, even if the variable's value happens to be `0`.

```
int zero = 0;
pi = zero; // error: cannot assign an int to a pointer
```

### ADVICE: INITIALIZE ALL POINTERS

Uninitialized pointers are a common source of run-time errors.

As with any other uninitialized variable, what happens when we use an uninitialized pointer is undefined. Using an uninitialized pointer almost always results in a run-time crash. However, debugging the resulting crashes can be surprisingly hard.

Under most compilers, when we use an uninitialized pointer, the bits in the memory in which the pointer resides are used as an address. Using an uninitialized pointer is a request to access a supposed object at that supposed location. There is no way to distinguish a valid address from an invalid one formed from the bits that happen to be in the memory in which the pointer was allocated.

Our recommendation to initialize all variables is particularly important for pointers. If possible, define a pointer only after the object to which it should point has been defined. If there is no object to bind to a pointer, then initialize the pointer to `nullptr` or `zero`. That way, the program can detect that the pointer does not point to an object.

## Assignment and Pointers

Both pointers and references give indirect access to other objects. However, there are important differences in how they do so. The most important is that a reference

is not an object. Once we have defined a reference, there is no way to make that reference refer to a different object. When we use a reference, we always get the object to which the reference was initially bound.

There is no such identity between a pointer and the address that it holds. As with any other (nonreference) variable, when we assign to a pointer, we give the pointer itself a new value. Assignment makes the pointer point to a different object:

```
int i = 42;
int *pi = 0;    // pi is initialized but addresses no object
int *pi2 = &i; // pi2 initialized to hold the address of i
int *pi3;       // if pi3 is defined inside a block, pi3 is uninitialized
pi3 = pi2;      // pi3 and pi2 address the same object, e.g., i
pi2 = 0;        // pi2 now addresses no object
```

It can be hard to keep straight whether an assignment changes the pointer or the object to which the pointer points. The important thing to keep in mind is that assignment changes its left-hand operand. When we write

```
pi = &ival; // value in pi is changed; pi now points to ival
```

we assign a new value to `pi`, which changes the address that `pi` holds. On the other hand, when we write

```
*pi = 0; // value in ival is changed; pi is unchanged
```

then `*pi` (i.e., the value to which `pi` points) is changed.

## Other Pointer Operations

So long as the pointer has a valid value, we can use a pointer in a condition. Just as when we use an arithmetic value in a condition (§ 2.1.2, p. 35), if the pointer is 0, then the condition is `false`:

```
int ival = 1024;
int *pi = 0;    // pi is a valid, null pointer
int *pi2 = &ival; // pi2 is a valid pointer that holds the address of ival
if (pi) // pi has value 0, so condition evaluates as false
    // ...
if (pi2) // pi2 points to ival, so it is not 0; the condition evaluates as true
    // ...
```

Any nonzero pointer evaluates as `true`

Given two valid pointers of the same type, we can compare them using the equality (`==`) or inequality (`!=`) operators. The result of these operators has type `bool`. Two pointers are equal if they hold the same address and unequal otherwise. Two pointers hold the same address (i.e., are equal) if they are both null, if they address the same object, or if they are both pointers one past the same object. Note that it is possible for a pointer to an object and a pointer one past the end of a different object to hold the same address. Such pointers will compare equal.

Because these operations use the value of the pointer, a pointer used in a condition or in a comparison must be a valid pointer. Using an invalid pointer as a condition or in a comparison is undefined.

§ 3.5.3 (p. 117) will cover additional pointer operations.

## void\* Pointers

The type `void*` is a special pointer type that can hold the address of any object. Like any other pointer, a `void*` pointer holds an address, but the type of the object at that address is unknown:

```
double obj = 3.14, *pd = &obj;
// ok: void* can hold the address value of any data pointer type
void *pv = &obj; // obj can be an object of any type
pv = pd;        // pv can hold a pointer to any type
```

There are only a limited number of things we can do with a `void*` pointer: We can compare it to another pointer, we can pass it to or return it from a function, and we can assign it to another `void*` pointer. We cannot use a `void*` to operate on the object it addresses—we don't know that object's type, and the type determines what operations we can perform on the object.

Generally, we use a `void*` pointer to deal with memory as memory, rather than using the pointer to access the object stored in that memory. We'll cover using `void*` pointers in this way in § 19.1.1 (p. 821). § 4.11.3 (p. 163) will show how we can retrieve the address stored in a `void*` pointer.

### EXERCISES SECTION 2.3.2

**Exercise 2.18:** Write code to change the value of a pointer. Write code to change the value to which the pointer points.

**Exercise 2.19:** Explain the key differences between pointers and references.

**Exercise 2.20:** What does the following program do?

```
int i = 42;
int *p1 = &i;
*p1 = *p1 * *p1;
```

**Exercise 2.21:** Explain each of the following definitions. Indicate whether any are illegal and, if so, why.

```
int i = 0;
(a) double* dp = &i; (b) int *ip = i; (c) int *p = &i;
```

**Exercise 2.22:** Assuming `p` is a pointer to `int`, explain the following code:

```
if (p) // ...
if (*p) // ...
```

**Exercise 2.23:** Given a pointer `p`, can you determine whether `p` points to a valid object? If so, how? If not, why not?

**Exercise 2.24:** Why is the initialization of `p` legal but that of `lp` illegal?

```
int i = 42;    void *p = &i;    long *lp = &i;
```

### 2.3.3 Understanding Compound Type Declarations



As we've seen, a variable definition consists of a base type and a list of declarators. Each declarator can relate its variable to the base type differently from the other declarators in the same definition. Thus, a single definition might define variables of different types:

```
// i is an int; p is a pointer to int; r is a reference to int
int i = 1024, *p = &i, &r = i;
```



Many programmers are confused by the interaction between the base type and the type modification that may be part of a declarator.

### Defining Multiple Variables



It is a common misconception to think that the type modifier (`*` or `&`) applies to all the variables defined in a single statement. Part of the problem arises because we can put whitespace between the type modifier and the name being declared:

```
int* p; // legal but might be misleading
```

We say that this definition might be misleading because it suggests that `int*` is the type of each variable declared in that statement. Despite appearances, the base type of this declaration is `int`, not `int*`. The `*` modifies the type of `p`. It says nothing about any other objects that might be declared in the same statement:

```
int* p1, p2; // p1 is a pointer to int; p2 is an int
```

There are two common styles used to define multiple variables with pointer or reference type. The first places the type modifier adjacent to the identifier:

```
int *p1, *p2; // both p1 and p2 are pointers to int
```

This style emphasizes that the variable has the indicated compound type.

The second places the type modifier with the type but defines only one variable per statement:

```
int* p1; // p1 is a pointer to int
int* p2; // p2 is a pointer to int
```

This style emphasizes that the declaration defines a compound type.



There is no single right way to define pointers or references. The important thing is to choose a style and use it consistently.

In this book we use the first style and place the `*` (or the `&`) with the variable name.

### Pointers to Pointers

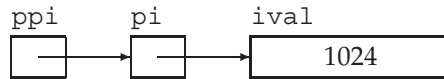
In general, there are no limits to how many type modifiers can be applied to a declarator. When there is more than one modifier, they combine in ways that are logical but not always obvious. As one example, consider a pointer. A pointer is

an object in memory, so like any object it has an address. Therefore, we can store the address of a pointer in another pointer.

We indicate each pointer level by its own `*`. That is, we write `**` for a pointer to a pointer, `***` for a pointer to a pointer to a pointer, and so on:

```
int ival = 1024;
int *pi = &ival;    // pi points to an int
int **ppi = &pi;    // ppi points to a pointer to an int
```

Here `pi` is a pointer to an `int` and `ppi` is a pointer to a pointer to an `int`. We might represent these objects as



Just as dereferencing a pointer to an `int` yields an `int`, dereferencing a pointer to a pointer yields a pointer. To access the underlying object, we must dereference the original pointer twice:

```
cout << "The value of ival\n"
    << "direct value: " << ival << "\n"
    << "indirect value: " << *pi << "\n"
    << "doubly indirect value: " << **ppi
    << endl;
```

This program prints the value of `ival` three different ways: first, directly; then, through the pointer to `int` in `pi`; and finally, by dereferencing `ppi` twice to get to the underlying value in `ival`.

## References to Pointers

A reference is not an object. Hence, we may not have a pointer to a reference. However, because a pointer is an object, we can define a reference to a pointer:

```
int i = 42;
int *p;          // p is a pointer to int
int *&r = p;     // r is a reference to the pointer p
r = &i;          // r refers to a pointer; assigning &i to r makes p point to i
*r = 0;          // dereferencing r yields i, the object to which p points; changes i to 0
```

The easiest way to understand the type of `r` is to read the definition right to left. The symbol closest to the name of the variable (in this case the `&` in `&r`) is the one that has the most immediate effect on the variable's type. Thus, we know that `r` is a reference. The rest of the declarator determines the type to which `r` refers. The next symbol, `*` in this case, says that the type `r` refers to is a pointer type. Finally, the base type of the declaration says that `r` is a reference to a pointer to an `int`.



It can be easier to understand complicated pointer or reference declarations if you read them from right to left.

**EXERCISES SECTION 2.3.3**

**Exercise 2.25:** Determine the types and values of each of the following variables.

(a) `int* ip, i, &r = i;` (b) `int i, *ip = 0;` (c) `int* ip, ip2;`

## 2.4 `const` Qualifier



Sometimes we want to define a variable whose value we know cannot be changed. For example, we might want to use a variable to refer to the size of a buffer size. Using a variable makes it easy for us to change the size of the buffer if we decided the original size wasn't what we needed. On the other hand, we'd also like to prevent code from inadvertently giving a new value to the variable we use to represent the buffer size. We can make a variable unchangeable by defining the variable's type as **`const`**:

```
const int bufSize = 512;    // input buffer size
```

defines `bufSize` as a constant. Any attempt to assign to `bufSize` is an error:

```
bufSize = 512; // error: attempt to write to const object
```

Because we can't change the value of a `const` object after we create it, it must be initialized. As usual, the initializer may be an arbitrarily complicated expression:

```
const int i = get_size(); // ok: initialized at run time
const int j = 42;         // ok: initialized at compile time
const int k;              // error: k is uninitialized const
```

### Initialization and `const`

As we have observed many times, the type of an object defines the operations that can be performed by that object. A `const` type can use most but not all of the same operations as its `nonconst` version. The one restriction is that we may use only those operations that cannot change an object. So, for example, we can use a `const int` in arithmetic expressions in exactly the same way as a plain, `nonconst int`. A `const int` converts to `bool` the same way as a plain `int`, and so on.

Among the operations that don't change the value of an object is initialization—when we use an object to initialize another object, it doesn't matter whether either or both of the objects are `const`s:

```
int i = 42;
const int ci = i;    // ok: the value in i is copied into ci
int j = ci;          // ok: the value in ci is copied into j
```

Although `ci` is a `const int`, the value in `ci` is an `int`. The `constness` of `ci` matters only for operations that might change `ci`. When we copy `ci` to initialize `j`, we don't care that `ci` is a `const`. Copying an object doesn't change that object. Once the copy is made, the new object has no further access to the original object.



## By Default, `const` Objects Are Local to a File

When a `const` object is initialized from a compile-time constant, such as in our definition of `bufSize`:

```
const int bufSize = 512;    // input buffer size
```

the compiler will usually replace uses of the variable with its corresponding value during compilation. That is, the compiler will generate code using the value 512 in the places that our code uses `bufSize`.

To substitute the value for the variable, the compiler has to see the variable's initializer. When we split a program into multiple files, every file that uses the `const` must have access to its initializer. In order to see the initializer, the variable must be defined in every file that wants to use the variable's value (§ 2.2.2, p. 45). To support this usage, yet avoid multiple definitions of the same variable, `const` variables are defined as local to the file. When we define a `const` with the same name in multiple files, it is as if we had written definitions for separate variables in each file.

Sometimes we have a `const` variable that we want to share across multiple files but whose initializer is not a constant expression. In this case, we don't want the compiler to generate a separate variable in each file. Instead, we want the `const` object to behave like other (nonconst) variables. We want to define the `const` in one file, and declare it in the other files that use that object.

To define a single instance of a `const` variable, we use the keyword `extern` on both its definition and declaration(s):

```
// file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();

// file_1.h
extern const int bufSize; // same bufSize as defined in file_1.cc
```

In this program, `file_1.cc` defines and initializes `bufSize`. Because this declaration includes an initializer, it is (as usual) a definition. However, because `bufSize` is `const`, we must specify `extern` in order for `bufSize` to be used in other files.

The declaration in `file_1.h` is also `extern`. In this case, the `extern` signifies that `bufSize` is not local to this file and that its definition will occur elsewhere.



To share a `const` object among multiple files, you must define the variable as `extern`.

## EXERCISES SECTION 2.4

**Exercise 2.26:** Which of the following are legal? For those that are illegal, explain why.

- |                                      |                               |
|--------------------------------------|-------------------------------|
| (a) <code>const int buf;</code>      | (b) <code>int cnt = 0;</code> |
| (c) <code>const int sz = cnt;</code> | (d) <code>++cnt; ++sz;</code> |

## 2.4.1 References to `const`



As with any other object, we can bind a reference to an object of a `const` type. To do so we use a **reference to `const`**, which is a reference that refers to a `const` type. Unlike an ordinary reference, a reference to `const` cannot be used to change the object to which the reference is bound:

```
const int ci = 1024;
const int &r1 = ci;    // ok: both reference and underlying object are const
r1 = 42;              // error: r1 is a reference to const
int &r2 = ci;          // error: nonconst reference to a const object
```

Because we cannot assign directly to `ci`, we also should not be able to use a reference to change `ci`. Therefore, the initialization of `r2` is an error. If this initialization were legal, we could use `r2` to change the value of its underlying object.

### TERMINOLOGY: `CONST` REFERENCE IS A REFERENCE TO `CONST`

C++ programmers tend to abbreviate the phrase “reference to `const`” as “`const` reference.” This abbreviation makes sense—if you remember that it is an abbreviation.

Technically speaking, there are no `const` references. A reference is not an object, so we cannot make a reference itself `const`. Indeed, because there is no way to make a reference refer to a different object, in some sense all references are `const`. Whether a reference refers to a `const` or `nonconst` type affects what we can do with that reference, not whether we can alter the binding of the reference itself.

## Initialization and References to `const`

In § 2.3.1 (p. 51) we noted that there are two exceptions to the rule that the type of a reference must match the type of the object to which it refers. The first exception is that we can initialize a reference to `const` from any expression that can be converted (§ 2.1.2, p. 35) to the type of the reference. In particular, we can bind a reference to `const` to a `nonconst` object, a literal, or a more general expression:

```
int i = 42;
const int &r1 = i;      // we can bind a const int& to a plain int object
const int &r2 = 42;     // ok: r2 is a reference to const
const int &r3 = r1 * 2; // ok: r3 is a reference to const
int &r4 = r1 * 2;       // error: r4 is a plain, nonconst reference
```

The easiest way to understand this difference in initialization rules is to consider what happens when we bind a reference to an object of a different type:

```
double dval = 3.14;
const int &ri = dval;
```

Here `ri` refers to an `int`. Operations on `ri` will be integer operations, but `dval` is a floating-point number, not an integer. To ensure that the object to which `ri` is bound is an `int`, the compiler transforms this code into something like

```
const int temp = dval;    // create a temporary const int from the double
const int &ri = temp;    // bind ri to that temporary
```

In this case, `ri` is bound to a **temporary** object. A temporary object is an unnamed object created by the compiler when it needs a place to store a result from evaluating an expression. C++ programmers often use the word *temporary* as an abbreviation for temporary object.

Now consider what could happen if this initialization were allowed but `ri` was not `const`. If `ri` weren't `const`, we could assign to `ri`. Doing so would change the object to which `ri` is bound. That object is a temporary, not `dval`. The programmer who made `ri` refer to `dval` would probably expect that assigning to `ri` would change `dval`. After all, why assign to `ri` unless the intent is to change the object to which `ri` is bound? Because binding a reference to a temporary is almost surely *not* what the programmer intended, the language makes it illegal.

## A Reference to `const` May Refer to an Object That Is Not `const`

It is important to realize that a reference to `const` restricts only what we can do through that reference. Binding a reference to `const` to an object says nothing about whether the underlying object itself is `const`. Because the underlying object might be `nonconst`, it might be changed by other means:

```
int i = 42;
int &r1 = i;          // r1 bound to i
const int &r2 = i;    // r2 also bound to i; but cannot be used to change i
r1 = 0;              // r1 is not const; i is now 0
r2 = 0;              // error: r2 is a reference to const
```

Binding `r2` to the (`nonconst`) `int i` is legal. However, we cannot use `r2` to change `i`. Even so, the value in `i` still might change. We can change `i` by assigning to it directly, or by assigning to another reference bound to `i`, such as `r1`.



## 2.4.2 Pointers and `const`

As with references, we can define pointers that point to either `const` or `nonconst` types. Like a reference to `const`, a **pointer to `const`** (§ 2.4.1, p. 61) may not be used to change the object to which the pointer points. We may store the address of a `const` object only in a pointer to `const`:

```
const double pi = 3.14;    // pi is const; its value may not be changed
double *ptr = &pi;         // error: ptr is a plain pointer
const double *cptr = &pi;  // ok: cptr may point to a double that is const
*cptr = 42;                // error: cannot assign to *cptr
```

In § 2.3.2 (p. 52) we noted that there are two exceptions to the rule that the types of a pointer and the object to which it points must match. The first exception is that we can use a pointer to `const` to point to a `nonconst` object:

```
double dval = 3.14;        // dval is a double; its value can be changed
cptr = &dval;               // ok: but can't change dval through cptr
```

Like a reference to `const`, a pointer to `const` says nothing about whether the object to which the pointer points is `const`. Defining a pointer as a pointer to `const` affects only what we can do with the pointer. It is important to remember that there is no guarantee that an object pointed to by a pointer to `const` won't change.



It may be helpful to think of pointers and references to `const` as pointers or references “that *think* they point or refer to `const`.”

## `const` Pointers

Unlike references, pointers are objects. Hence, as with any other object type, we can have a pointer that is itself `const`. Like any other `const` object, a **`const pointer`** must be initialized, and once initialized, its value (i.e., the address that it holds) may not be changed. We indicate that the pointer is `const` by putting the `const` after the `*`. This placement indicates that it is the pointer, not the pointed-to type, that is `const`:

```
int errNumb = 0;
int *const curErr = &errNumb; // curErr will always point to errNumb
const double pi = 3.14159;
const double *const pip = &pi; // pip is a const pointer to a const object
```

As we saw in § 2.3.3 (p. 58), the easiest way to understand these declarations is to read them from right to left. In this case, the symbol closest to `curErr` is `const`, which means that `curErr` itself will be a `const` object. The type of that object is formed from the rest of the declarator. The next symbol in the declarator is `*`, which means that `curErr` is a `const` pointer. Finally, the base type of the declaration completes the type of `curErr`, which is a `const` pointer to an object of type `int`. Similarly, `pip` is a `const` pointer to an object of type `const double`.

The fact that a pointer is itself `const` says nothing about whether we can use the pointer to change the underlying object. Whether we can change that object depends entirely on the type to which the pointer points. For example, `pip` is a `const` pointer to `const`. Neither the value of the object addressed by `pip` nor the address stored in `pip` can be changed. On the other hand, `curErr` addresses a plain, nonconst `int`. We can use `curErr` to change the value of `errNumb`:

```
*pip = 2.72; // error: pip is a pointer to const
// if the object to which curErr points (i.e., errNumb) is nonzero
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset the value of the object to which curErr is bound
}
```

### 2.4.3 Top-Level `const`



As we've seen, a pointer is an object that can point to a different object. As a result, we can talk independently about whether a pointer is `const` and whether

## EXERCISES SECTION 2.4.2

**Exercise 2.27:** Which of the following initializations are legal? Explain why.

- (a) `int i = -1, &r = 0;`                      (b) `int *const p2 = &i2;`  
 (c) `const int i = -1, &r = 0;`                (d) `const int *const p3 = &i2;`  
 (e) `const int *p1 = &i2;`                      (f) `const int &const r2;`  
 (g) `const int i2 = i, &r = i;`

**Exercise 2.28:** Explain the following definitions. Identify any that are illegal.

- (a) `int i, *const cp;`                      (b) `int *p1, *const p2;`  
 (c) `const int ic, &r = ic;`                (d) `const int *const p3;`  
 (e) `const int *p;`

**Exercise 2.29:** Using the variables in the previous exercise, which of the following assignments are legal? Explain why.

- (a) `i = ic;`                                      (b) `p1 = p3;`  
 (c) `p1 = &ic;`                                  (d) `p3 = &ic;`  
 (e) `p2 = p1;`                                  (f) `ic = *p3;`

the objects to which it can point are `const`. We use the term **top-level `const`** to indicate that the pointer itself is a `const`. When a pointer can point to a `const` object, we refer to that `const` as a **low-level `const`**.

More generally, top-level `const` indicates that an object itself is `const`. Top-level `const` can appear in any object type, i.e., one of the built-in arithmetic types, a class type, or a pointer type. Low-level `const` appears in the base type of compound types such as pointers or references. Note that pointer types, unlike most other types, can have both top-level and low-level `const` independently:

```
int i = 0;
int *const p1 = &i; // we can't change the value of p1; const is top-level
const int ci = 42; // we cannot change ci; const is top-level
const int *p2 = &ci; // we can change p2; const is low-level
const int *const p3 = p2; // right-most const is top-level, left-most is not
const int &r = ci; // const in reference types is always low-level
```



The distinction between top-level and low-level matters when we copy an object. When we copy an object, top-level `const`s are ignored:

```
i = ci; // ok: copying the value of ci; top-level const in ci is ignored
p2 = p3; // ok: pointed-to type matches; top-level const in p3 is ignored
```

Copying an object doesn't change the copied object. As a result, it is immaterial whether the object copied from or copied into is `const`.

On the other hand, low-level `const` is never ignored. When we copy an object, both objects must have the same low-level `const` qualification or there must be a conversion between the types of the two objects. In general, we can convert a `nonconst` to `const` but not the other way round:

```

int *p = p3; // error: p3 has a low-level const but p doesn't
p2 = p3;     // ok: p2 has the same low-level const qualification as p3
p2 = &i;     // ok: we can convert int* to const int*
int &r = ci;  // error: can't bind an ordinary int& to a const int object
const int &r2 = i; // ok: can bind const int& to plain int

```

`p3` has both a top-level and low-level `const`. When we copy `p3`, we can ignore its top-level `const` but not the fact that it points to a `const` type. Hence, we cannot use `p3` to initialize `p`, which points to a plain (nonconst) `int`. On the other hand, we can assign `p3` to `p2`. Both pointers have the same (low-level `const`) type. The fact that `p3` is a `const` pointer (i.e., that it has a top-level `const`) doesn't matter.

### EXERCISES SECTION 2.4.3

**Exercise 2.30:** For each of the following declarations indicate whether the object being declared has top-level or low-level `const`.

```

const int v2 = 0;    int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;

```

**Exercise 2.31:** Given the declarations in the previous exercise determine whether the following assignments are legal. Explain how the top-level or low-level `const` applies in each case.

```

r1 = v2;
p1 = p2;    p2 = p1;
p1 = p3;    p2 = p3;

```

## 2.4.4 `constexpr` and Constant Expressions



A **constant expression** is an expression whose value cannot change and that can be evaluated at compile time. A literal is a constant expression. A `const` object that is initialized from a constant expression is also a constant expression. As we'll see, there are several contexts in the language that require constant expressions.

Whether a given object (or expression) is a constant expression depends on the types and the initializers. For example:

```

const int max_files = 20; // max_files is a constant expression
const int limit = max_files + 1; // limit is a constant expression
int staff_size = 27; // staff_size is not a constant expression
const int sz = get_size(); // sz is not a constant expression

```

Although `staff_size` is initialized from a literal, it is not a constant expression because it is a plain `int`, not a `const int`. On the other hand, even though `sz` is a `const`, the value of its initializer is not known until run time. Hence, `sz` is not a constant expression.

## constexpr Variables

In a large system, it can be difficult to determine (for certain) that an initializer is a constant expression. We might define a `const` variable with an initializer that we think is a constant expression. However, when we use that variable in a context that requires a constant expression we may discover that the initializer was not a constant expression. In general, the definition of an object and its use in such a context can be widely separated.

C++  
11

Under the new standard, we can ask the compiler to verify that a variable is a constant expression by declaring the variable in a **constexpr** declaration. Variables declared as `constexpr` are implicitly `const` and must be initialized by constant expressions:

```
constexpr int mf = 20;           // 20 is a constant expression
constexpr int limit = mf + 1;    // mf + 1 is a constant expression
constexpr int sz = size();       // ok only if size is a constexpr function
```

Although we cannot use an ordinary function as an initializer for a `constexpr` variable, we'll see in § 6.5.2 (p. 239) that the new standard lets us define certain functions as `constexpr`. Such functions must be simple enough that the compiler can evaluate them at compile time. We can use `constexpr` functions in the initializer of a `constexpr` variable.



Generally, it is a good idea to use `constexpr` for variables that you intend to use as constant expressions.

## Literal Types

Because a constant expression is one that can be evaluated at compile time, there are limits on the types that we can use in a `constexpr` declaration. The types we can use in a `constexpr` are known as “literal types” because they are simple enough to have literal values.

Of the types we have used so far, the arithmetic, reference, and pointer types are literal types. Our `Sales_item` class and the library `IO` and `string` types are not literal types. Hence, we cannot define variables of these types as `constexpr`s. We'll see other kinds of literal types in § 7.5.6 (p. 299) and § 19.3 (p. 832).

Although we can define both pointers and reference as `constexpr`s, the objects we use to initialize them are strictly limited. We can initialize a `constexpr` pointer from the `nullptr` literal or the literal (i.e., constant expression) `0`. We can also point to (or bind to) an object that remains at a fixed address.

For reasons we'll cover in § 6.1.1 (p. 204), variables defined inside a function ordinarily are not stored at a fixed address. Hence, we cannot use a `constexpr` pointer to point to such variables. On the other hand, the address of an object defined outside of any function is a constant expression, and so may be used to initialize a `constexpr` pointer. We'll see in § 6.1.1 (p. 205), that functions may define variables that exist across calls to that function. Like an object defined outside any function, these special local objects also have fixed addresses. Therefore, a `constexpr` reference may be bound to, and a `constexpr` pointer may address, such variables.

## Pointers and `constexpr`

It is important to understand that when we define a pointer in a `constexpr` declaration, the `constexpr` specifier applies to the pointer, not the type to which the pointer points:

```
const int *p = nullptr;    // p is a pointer to a const int
constexpr int *q = nullptr; // q is a const pointer to int
```

Despite appearances, the types of `p` and `q` are quite different; `p` is a pointer to `const`, whereas `q` is a constant pointer. The difference is a consequence of the fact that `constexpr` imposes a top-level `const` (§ 2.4.3, p. 63) on the objects it defines.

Like any other constant pointer, a `constexpr` pointer may point to a `const` or a `nonconst` type:

```
constexpr int *np = nullptr; // np is a constant pointer to int that is null
int j = 0;
constexpr int i = 42; // type of i is const int
// i and j must be defined outside any function
constexpr const int *p = &i; // p is a constant pointer to the const int i
constexpr int *p1 = &j; // p1 is a constant pointer to the int j
```

### EXERCISES SECTION 2.4.4

**Exercise 2.32:** Is the following code legal or not? If not, how might you make it legal?

```
int null = 0, *p = null;
```

## 2.5 Dealing with Types

As our programs get more complicated, we'll see that the types we use also get more complicated. Complications in using types arise in two different ways. Some types are hard to "spell." That is, they have forms that are tedious and error-prone to write. Moreover, the form of a complicated type can obscure its purpose or meaning. The other source of complication is that sometimes it is hard to determine the exact type we need. Doing so can require us to look back into the context of the program.

### 2.5.1 Type Aliases

A **type alias** is a name that is a synonym for another type. Type aliases let us simplify complicated type definitions, making those types easier to use. Type aliases also let us emphasize the purpose for which a type is used.

We can define a type alias in one of two ways. Traditionally, we use a **typedef**:

```
typedef double wages; // wages is a synonym for double
typedef wages base, *p; // base is a synonym for double, p for double*
```



The keyword `typedef` may appear as part of the base type of a declaration (§ 2.3, p. 50). Declarations that include `typedef` define type aliases rather than variables. As in any other declaration, the declarators can include type modifiers that define compound types built from the base type of the definition.



The new standard introduced a second way to define a type alias, via an **alias declaration**:

```
using SI = Sales_item;    // SI is a synonym for Sales_item
```

An alias declaration starts with the keyword `using` followed by the alias name and an `=`. The alias declaration defines the name on the left-hand side of the `=` as an alias for the type that appears on the right-hand side.

A type alias is a type name and can appear wherever a type name can appear:

```
wages hourly, weekly;    // same as double hourly, weekly;
SI item;                  // same as Sales_item item
```



## Pointers, `const`, and Type Aliases

Declarations that use type aliases that represent compound types and `const` can yield surprising results. For example, the following declarations use the type `pstring`, which is an alias for the type `char*`:

```
typedef char *pstring;
const pstring cstr = 0; // cstr is a constant pointer to char
const pstring *ps;     // ps is a pointer to a constant pointer to char
```

The base type in these declarations is `const pstring`. As usual, a `const` that appears in the base type modifies the given type. The type of `pstring` is “pointer to `char`.” So, `const pstring` is a constant pointer to `char`—not a pointer to `const char`.

It can be tempting, albeit incorrect, to interpret a declaration that uses a type alias by conceptually replacing the alias with its corresponding type:

```
const char *cstr = 0; // wrong interpretation of const pstring cstr
```

However, this interpretation is wrong. When we use `pstring` in a declaration, the base type of the declaration is a pointer type. When we rewrite the declaration using `char*`, the base type is `char` and the `*` is part of the declarator. In this case, `const char` is the base type. This rewrite declares `cstr` as a pointer to `const char` rather than as a `const` pointer to `char`.



## 2.5.2 The `auto` Type Specifier

It is not uncommon to want to store the value of an expression in a variable. To declare the variable, we have to know the type of that expression. When we write a program, it can be surprisingly difficult—and sometimes even impossible—to determine the type of an expression. Under the new standard, we can let the compiler figure out the type for us by using the **`auto`** type specifier. Unlike type specifiers, such as `double`, that name a specific type, `auto` tells the compiler to deduce



the type from the initializer. By implication, a variable that uses `auto` as its type specifier must have an initializer:

```
// the type of item is deduced from the type of the result of adding val1 and val2
auto item = val1 + val2; // item initialized to the result of val1 + val2
```

Here the compiler will deduce the type of `item` from the type returned by applying `+` to `val1` and `val2`. If `val1` and `val2` are `Sales_item` objects (§ 1.5, p. 19), `item` will have type `Sales_item`. If those variables are type `double`, then `item` has type `double`, and so on.

As with any other type specifier, we can define multiple variables using `auto`. Because a declaration can involve only a single base type, the initializers for all the variables in the declaration must have types that are consistent with each other:

```
auto i = 0, *p = &i; // ok: i is int and p is a pointer to int
auto sz = 0, pi = 3.14; // error: inconsistent types for sz and pi
```

## Compound Types, `const`, and `auto`

The type that the compiler infers for `auto` is not always exactly the same as the initializer's type. Instead, the compiler adjusts the type to conform to normal initialization rules.

First, as we've seen, when we use a reference, we are really using the object to which the reference refers. In particular, when we use a reference as an initializer, the initializer is the corresponding object. The compiler uses that object's type for `auto`'s type deduction:

```
int i = 0, &r = i;
auto a = r; // a is an int (r is an alias for i, which has type int)
```

Second, `auto` ordinarily ignores top-level `const`s (§ 2.4.3, p. 63). As usual in initializations, low-level `const`s, such as when an initializer is a pointer to `const`, are kept:

```
const int ci = i, &cr = ci;
auto b = ci; // b is an int (top-level const in ci is dropped)
auto c = cr; // c is an int (cr is an alias for ci whose const is top-level)
auto d = &i; // d is an int* (& of an int object is int*)
auto e = &ci; // e is const int* (& of a const object is low-level const)
```

If we want the deduced type to have a top-level `const`, we must say so explicitly:

```
const auto f = ci; // deduced type of ci is int; f has type const int
```

We can also specify that we want a reference to the `auto`-deduced type. Normal initialization rules still apply:

```
auto &g = ci; // g is a const int& that is bound to ci
auto &h = 42; // error: we can't bind a plain reference to a literal
const auto &j = 42; // ok: we can bind a const reference to a literal
```