

# Overview

	Introduction	1
1	The Big Picture	9
2	C++ Programming Basics	29
3	Loops and Decisions	75
4	Structures	131
5	Functions	161
6	Objects and Classes	215
7	Arrays and Strings	263
8	Operator Overloading	319
9	Inheritance	371
10	Pointers	429
11	Virtual Functions	503
12	Streams and Files	567
13	Multifile Programs	633
14	Templates and Exceptions	681
15	The Standard Template Library	725
16	Object-Oriented Software Development	801
A	ASCII Chart	849
B	C++ Precedence Table and Keywords	859
C	Microsoft Visual C++	863
D	Borland C++Builder	871
E	Console Graphics Lite	881
F	STL Algorithms and Member Functions	895
G	Answers to Questions and Exercises	913
H	Bibliography	977
	Index	981

# Contents

## Introduction 1

### 1 The Big Picture 9

Why Do We Need Object-Oriented Programming? .....	10
Procedural Languages .....	10
The Object-Oriented Approach .....	13
Characteristics of Object-Oriented Languages .....	16
Objects .....	16
Classes .....	18
Inheritance .....	18
Reusability .....	21
Creating New Data Types .....	21
Polymorphism and Overloading .....	21
C++ and C .....	22
Laying the Groundwork .....	23
The Unified Modeling Language (UML) .....	23
Summary .....	25
Questions .....	25

### 2 C++ Programming Basics 29

Getting Started .....	30
Basic Program Construction .....	30
Functions .....	31
Program Statements .....	32
Whitespace .....	33
Output Using cout .....	33
String Constants .....	34
Directives .....	35
Preprocessor Directives .....	35
Header Files .....	35
The using Directive .....	36
Comments .....	36
Comment Syntax .....	36
When to Use Comments .....	37
Alternative Comment Syntax .....	37
Integer Variables .....	38
Defining Integer Variables .....	38
Declarations and Definitions .....	40
Variable Names .....	40
Assignment Statements .....	40

Integer Constants .....	41
Output Variations .....	41
The endl Manipulator .....	41
Other Integer Types .....	42
Character Variables .....	42
Character Constants .....	43
Initialization .....	44
Escape Sequences .....	44
Input with cin .....	45
Variables Defined at Point of Use .....	47
Cascading << .....	47
Expressions .....	47
Precedence .....	47
Floating Point Types .....	48
Type float .....	48
Type double and long double .....	49
Floating-Point Constants .....	50
The const Qualifier .....	51
The #define Directive .....	51
Type bool .....	51
The setw Manipulator .....	52
Cascading the Insertion Operator .....	54
Multiple Definitions .....	54
The iomanip Header File .....	54
Variable Type Summary .....	54
unsigned Data Types .....	55
Type Conversion .....	56
Automatic Conversions .....	57
Casts .....	58
Arithmetic Operators .....	60
The Remainder Operator .....	61
Arithmetic Assignment Operators .....	61
Increment Operators .....	63
Library Functions .....	65
Header Files .....	66
Library Files .....	66
Header Files and Library Files .....	67
Two Ways to Use #include .....	67
Summary .....	68
Questions .....	69
Exercises .....	71

<b>3</b>	<b>Loops and Decisions</b>	<b>75</b>
	Relational Operators .....	76
	Loops.....	78
	The for Loop.....	78
	Debugging Animation .....	84
	for Loop Variations .....	84
	The while Loop .....	86
	Precedence: Arithmetic and Relational Operators .....	89
	The do Loop.....	91
	When to Use Which Loop .....	93
	Decisions.....	93
	The if Statement.....	94
	The if...else Statement .....	98
	The else...if Construction .....	106
	The switch Statement .....	107
	The Conditional Operator .....	111
	Logical Operators .....	114
	Logical AND Operator .....	115
	Logical OR Operator.....	116
	Logical NOT Operator .....	117
	Precedence Summary .....	118
	Other Control Statements .....	118
	The break Statement .....	119
	The continue Statement .....	121
	The goto Statement .....	123
	Summary.....	123
	Questions.....	124
	Exercises .....	126
<b>4</b>	<b>Structures</b>	<b>131</b>
	Structures .....	132
	A Simple Structure .....	132
	Defining the Structure .....	133
	Defining a Structure Variable .....	134
	Accessing Structure Members.....	136
	Other Structure Features .....	137
	A Measurement Example .....	139
	Structures Within Structures .....	141
	A Card Game Example .....	145
	Structures and Classes.....	148
	Enumerations .....	148
	Days of the Week.....	148
	One Thing or Another .....	151

Organizing the Cards .....	153
Specifying Integer Values .....	155
Not Perfect.....	155
Other Examples .....	155
Summary .....	156
Questions.....	156
Exercises .....	158
<b>5 Functions 161</b>	
Simple Functions .....	162
The Function Declaration .....	164
Calling the Function .....	164
The Function Definition .....	164
Comparison with Library Functions .....	166
Eliminating the Declaration.....	166
Passing Arguments to Functions.....	167
Passing Constants .....	167
Passing Variables .....	169
Passing by Value .....	170
Structures as Arguments .....	171
Names in the Declaration .....	176
Returning Values from Functions .....	176
The return Statement .....	177
Returning Structure Variables .....	180
Reference Arguments .....	182
Passing Simple Data Types by Reference .....	182
A More Complex Pass by Reference .....	185
Passing Structures by Reference .....	186
Notes on Passing by Reference .....	188
Overloaded Functions .....	188
Different Numbers of Arguments .....	189
Different Kinds of Arguments .....	191
Recursion .....	193
Inline Functions .....	195
Default Arguments .....	197
Scope and Storage Class.....	199
Local Variables .....	199
Global Variables.....	202
Static Local Variables .....	204
Storage.....	205
Returning by Reference .....	206
Function Calls on the Left of the Equal Sign .....	207
Don't Worry Yet.....	207

const Function Arguments .....	208
Summary .....	209
Questions.....	210
Exercises .....	212
<b>6 Objects and Classes 215</b>	
A Simple Class .....	216
Classes and Objects .....	217
Defining the Class .....	218
Using the Class .....	221
Calling Member Functions .....	221
C++ Objects as Physical Objects .....	223
Widget Parts as Objects.....	223
Circles as Objects .....	224
C++ Objects as Data Types .....	226
Constructors .....	227
A Counter Example .....	228
A Graphics Example .....	231
Destructors.....	232
Objects as Function Arguments .....	233
Overloaded Constructors .....	234
Member Functions Defined Outside the Class .....	236
Objects as Arguments .....	237
The Default Copy Constructor .....	238
Returning Objects from Functions .....	240
Arguments and Objects .....	241
A Card-Game Example.....	243
Structures and Classes .....	247
Classes, Objects, and Memory .....	247
Static Class Data .....	249
Uses of Static Class Data .....	249
An Example of Static Class Data .....	249
Separate Declaration and Definition .....	251
const and Classes .....	252
const Member Functions .....	252
const Objects .....	255
What Does It All Mean? .....	256
Summary .....	257
Questions.....	257
Exercises .....	259

**7 Arrays and Strings 263**

Array Fundamentals .....	264
Defining Arrays .....	265
Array Elements .....	265
Accessing Array Elements.....	267
Averaging Array Elements .....	267
Initializing Arrays .....	268
Multidimensional Arrays .....	270
Passing Arrays to Functions .....	274
Arrays of Structures.....	277
Arrays as Class Member Data .....	279
Arrays of Objects .....	283
Arrays of English Distances .....	283
Arrays of Cards .....	286
C-Strings .....	290
C-String Variables .....	290
Avoiding Buffer Overflow.....	292
String Constants.....	292
Reading Embedded Blanks .....	293
Reading Multiple Lines .....	294
Copying a String the Hard Way .....	295
Copying a String the Easy Way.....	296
Arrays of Strings .....	297
Strings as Class Members .....	298
A User-Defined String Type .....	300
The Standard C++ string Class.....	302
Defining and Assigning string Objects .....	302
Input/Output with string Objects.....	304
Finding string Objects .....	305
Modifying string Objects .....	306
Comparing string Objects .....	307
Accessing Characters in string Objects.....	309
Other string Functions.....	310
Summary .....	310
Questions.....	311
Exercises .....	313

**8 Operator Overloading 319**

Overloading Unary Operators.....	320
The operator Keyword .....	322
Operator Arguments .....	323

Operator Return Values .....	323
Nameless Temporary Objects .....	325
Postfix Notation .....	326
Overloading Binary Operators .....	328
Arithmetic Operators .....	328
Concatenating Strings .....	332
Multiple Overloading .....	334
Comparison Operators .....	334
Arithmetic Assignment Operators .....	337
The Subscript Operator ([]) .....	340
Data Conversion .....	344
Conversions Between Basic Types .....	344
Conversions Between Objects and Basic Types .....	345
Conversions Between Objects of Different Classes .....	350
Conversions: When to Use What .....	357
UML Class Diagrams .....	357
Associations .....	357
Navigability .....	358
Pitfalls of Operator Overloading and Conversion .....	358
Use Similar Meanings .....	358
Use Similar Syntax .....	359
Show Restraint .....	359
Avoid Ambiguity .....	360
Not All Operators Can Be Overloaded .....	360
Keywords <code>explicit</code> and <code>mutable</code> .....	360
Preventing Conversions with <code>explicit</code> .....	360
Changing <code>const</code> Object Data Using <code>mutable</code> .....	362
Summary .....	364
Questions .....	364
Exercises .....	367
<b>9 Inheritance 371</b>	
Derived Class and Base Class .....	373
Specifying the Derived Class .....	375
Generalization in UML Class Diagrams .....	375
Accessing Base Class Members .....	376
The <code>protected</code> Access Specifier .....	377
Derived Class Constructors .....	380
Overriding Member Functions .....	382
Which Function Is Used? .....	383
Scope Resolution with Overridden Functions .....	384



Inheritance in the English Distance Class .....	384
Operation of ENGLN .....	387
Constructors in DistSign .....	387
Member Functions in DistSign .....	387
Abetting Inheritance .....	388
Class Hierarchies .....	388
“Abstract” Base Class .....	392
Constructors and Member Functions .....	393
Inheritance and Graphics Shapes .....	393
Public and Private Inheritance .....	396
Access Combinations .....	397
Access Specifiers: When to Use What .....	399
Levels of Inheritance .....	399
Multiple Inheritance .....	403
Member Functions in Multiple Inheritance.....	404
private Derivation in EMPMULT .....	409
Constructors in Multiple Inheritance.....	409
Ambiguity in Multiple Inheritance .....	413
Aggregation: Classes Within Classes .....	414
Aggregation in the EMPCONT Program.....	416
Composition: A Stronger Aggregation .....	420
Inheritance and Program Development .....	420
Summary .....	421
Questions.....	422
Exercises .....	424
<b>10 Pointers 429</b>	
Addresses and Pointers .....	430
The Address-of Operator & .....	431
Pointer Variables .....	433
Syntax Quibbles.....	434
Accessing the Variable Pointed To .....	436
Pointer to void.....	439
Pointers and Arrays.....	440
Pointer Constants and Pointer Variables .....	442
Pointers and Functions .....	443
Passing Simple Variables.....	443
Passing Arrays .....	446
Sorting Array Elements .....	448
Pointers and C-Type Strings .....	452
Pointers to String Constants .....	452
Strings as Function Arguments .....	453

Copying a String Using Pointers .....	454
Library String Functions .....	456
The const Modifier and Pointers .....	456
Arrays of Pointers to Strings .....	456
Memory Management: new and delete .....	458
The new Operator .....	459
The delete Operator .....	461
A String Class Using new .....	462
Pointers to Objects .....	464
Referring to Members .....	465
Another Approach to new .....	465
An Array of Pointers to Objects .....	467
A Linked List Example .....	469
A Chain of Pointers .....	469
Adding an Item to the List .....	471
Displaying the List Contents .....	472
Self-Containing Classes .....	473
Augmenting LINKLIST .....	473
Pointers to Pointers .....	474
Sorting Pointers .....	476
The person** Data Type .....	476
Comparing Strings .....	478
A Parsing Example .....	479
Parsing Arithmetic Expressions .....	479
The PARSE Program .....	481
Simulation: A Horse Race .....	484
Designing the Horse Race .....	485
Multiplicity in the UML .....	489
UML State Diagrams .....	490
States .....	491
Transitions .....	491
Racing from State to State .....	492
Debugging Pointers .....	492
Summary .....	493
Questions .....	494
Exercises .....	497
 <b>11 Virtual Functions   503</b>	
Virtual Functions .....	504
Normal Member Functions Accessed with Pointers .....	505
Virtual Member Functions Accessed with Pointers .....	507
Late Binding .....	509

Abstract Classes and Pure Virtual Functions .....	510
Virtual Functions and the person Class .....	511
Virtual Functions in a Graphics Example .....	514
Virtual Destructors.....	517
Virtual Base Classes .....	518
Friend Functions .....	520
Friends as Bridges .....	520
Breaching the Walls.....	522
English Distance Example.....	522
friends for Functional Notation .....	526
friend Classes.....	528
Static Functions.....	529
Accessing static Functions .....	531
Numbering the Objects .....	532
Investigating Destructors .....	532
Assignment and Copy Initialization .....	532
Overloading the Assignment Operator .....	533
The Copy Constructor .....	536
UML Object Diagrams .....	539
A Memory-Efficient String Class .....	540
The this Pointer.....	547
Accessing Member Data with this.....	547
Using this for Returning Values.....	548
Revised STRIMEM Program.....	550
Dynamic Type Information.....	553
Checking the Type of a Class with dynamic_cast .....	553
Changing Pointer Types with dynamic_cast.....	554
The typeid Operator .....	556
Summary.....	557
Questions.....	558
Exercises .....	561
<b>12 Streams and Files   567</b>	
Stream Classes .....	568
Advantages of Streams .....	568
The Stream Class Hierarchy .....	568
The ios Class .....	570
The istream Class.....	574
The ostream Class.....	575
The iostream and the _withassign Classes.....	576
Stream Errors .....	577
Error-Status Bits .....	577
Inputting Numbers.....	578

Too Many Characters .....	579
No-Input Input .....	579
Inputting Strings and Characters .....	580
Error-Free Distances .....	580
Disk File I/O with Streams .....	583
Formatted File I/O .....	583
Strings with Embedded Blanks .....	586
Character I/O .....	588
Binary I/O .....	589
The reinterpret_cast Operator .....	591
Closing Files .....	591
Object I/O .....	591
I/O with Multiple Objects .....	594
File Pointers .....	597
Specifying the Position .....	598
Specifying the Offset .....	598
The tellg() Function .....	601
Error Handling in File I/O .....	601
Reacting to Errors .....	601
Analyzing Errors .....	602
File I/O with Member Functions .....	604
Objects That Read and Write Themselves .....	604
Classes That Read and Write Themselves .....	607
Overloading the Extraction and Insertion Operators .....	616
Overloading for cout and cin .....	616
Overloading for Files .....	618
Memory as a Stream Object .....	620
Command-Line Arguments .....	622
Printer Output .....	624
Summary .....	626
Questions .....	627
Exercises .....	628
<b>13 Multifile Programs 633</b>	
Reasons for Multifile Programs .....	634
Class Libraries .....	634
Organization and Conceptualization .....	635
Creating a Multifile Program .....	637
Header Files .....	637
Directory .....	637
Projects .....	637

Inter-File Communication.....	638
Communication Among Source Files .....	638
Header Files.....	643
Namespaces .....	647
A Very Long Number Class .....	651
Numbers as Strings .....	652
The Class Specifier .....	652
The Member Functions .....	654
The Application Program .....	657
A High-Rise Elevator Simulation .....	658
Running the ELEV Program .....	658
Designing the System .....	660
Listings for ELEV .....	662
Elevator Strategy .....	674
State Diagram for the ELEV Program.....	675
Summary.....	676
Questions.....	677
Projects .....	679
 <b>14 Templates and Exceptions 681</b>	
Function Templates.....	682
A Simple Function Template.....	684
Function Templates with Multiple Arguments .....	686
Class Templates .....	690
Class Name Depends on Context .....	694
A Linked List Class Using Templates.....	696
Storing User-Defined Data Types .....	698
The UML and Templates.....	702
Exceptions.....	703
Why Do We Need Exceptions? .....	703
Exception Syntax.....	704
A Simple Exception Example .....	706
Multiple Exceptions.....	710
Exceptions with the Distance Class .....	712
Exceptions with Arguments.....	714
The bad_alloc Class .....	717
Exception Notes.....	718
Summary.....	720
Questions.....	720
Exercises .....	722

**15 The Standard Template Library 725**

Introduction to the STL .....	726
Containers .....	727
Algorithms .....	732
Iterators .....	733
Potential Problems with the STL .....	734
Algorithms .....	735
The find() Algorithm .....	735
The count() Algorithm .....	736
The sort() Algorithm .....	737
The search() Algorithm .....	737
The merge() Algorithm .....	738
Function Objects .....	739
The for_each() Algorithm .....	742
The transform() Algorithm .....	742
Sequence Containers.....	743
Vectors .....	743
Lists .....	747
Deque .....	750
Iterators .....	751
Iterators as Smart Pointers.....	752
Iterators as an Interface .....	753
Matching Algorithms with Containers .....	755
Iterators at Work .....	759
Specialized Iterators .....	763
Iterator Adapters .....	763
Stream Iterators .....	767
Associative Containers .....	771
Sets and Multisets .....	771
Maps and Multimaps .....	775
Storing User-Defined Objects.....	778
A Set of person Objects .....	778
A List of person Objects .....	782
Function Objects .....	786
Predefined Function Objects .....	786
Writing Your Own Function Objects.....	789
Function Objects Used to Modify Container Behavior .....	794
Summary .....	794
Questions.....	795
Exercises .....	797

<b>16 Object-Oriented Software Development</b>	<b>801</b>
Evolution of the Software Development Processes .....	802
The Seat-of-the-Pants Process .....	802
The Waterfall Process .....	802
Object-Oriented Programming .....	803
Modern Processes .....	803
Use Case Modeling .....	805
Actors .....	805
Use Cases .....	806
Scenarios .....	806
Use Case Diagrams .....	806
Use Case Descriptions .....	807
From Use Cases to Classes .....	808
The Programming Problem .....	809
Hand-Written Forms .....	809
Assumptions .....	811
The Elaboration Phase for the LANDLORD Program .....	812
Actors .....	812
Use Cases .....	812
Use Case Descriptions .....	813
Scenarios .....	815
UML Activity Diagrams .....	815
From Use Cases to Classes .....	816
Listing the Nouns .....	816
Refining the List .....	817
Discovering Attributes .....	818
From Verbs to Messages .....	818
Class Diagram .....	820
Sequence Diagrams .....	820
Writing the Code .....	824
The Header File .....	825
The .CPP Files .....	831
More Simplifications .....	841
Interacting with the Program .....	841
Final Thoughts .....	843
Summary .....	844
Questions .....	844
Projects .....	846
<b>A ASCII Chart</b>	<b>849</b>
<b>B C++ Precedence Table and Keywords</b>	<b>859</b>
Precedence Table .....	860
Keywords .....	860

**C Microsoft Visual C++ 863**

Screen Elements .....	864
Single-File Programs .....	864
Building an Existing File.....	864
Writing a New File .....	865
Errors .....	865
Run-Time Type Information (RTTI) .....	866
Multifile Programs .....	866
Projects and Workspaces .....	866
Developing the Project .....	867
Saving, Closing, and Opening Projects.....	868
Compiling and Linking .....	868
Building Console Graphics Lite Programs.....	868
Debugging.....	868
Single-Stepping .....	869
Watching Variables .....	869
Stepping Into Functions.....	869
Breakpoints .....	870

**D Borland C++Builder 871**

Running the Example Programs in C++Builder.....	872
Cleaning Up the Screen .....	873
Creating a New Project.....	873
Naming and Saving a Project .....	874
Starting with Existing Files .....	875
Compiling, Linking, and Executing .....	875
Executing from C++Builder .....	875
Executing from MS-DOS .....	875
Precompiled Header Files .....	876
Closing and Opening Projects.....	876
Adding a Header File to Your Project .....	876
Creating a New Header File .....	876
Editing an Existing Header File .....	876
Telling C++Builder the Header File's Location .....	877
Projects with Multiple Source Files .....	877
Creating Additional Source Files .....	877
Adding Existing Source Files to Your Project .....	877
The Project Manager .....	878
Console Graphics Lite Programs .....	878
Debugging.....	878
Single-Stepping .....	879
Watching Variables .....	879
Tracing into Functions.....	879
Breakpoints .....	879



<b>E</b>	<b>Console Graphics Lite</b>	<b>881</b>
	Using the Console Graphics Lite Routines .....	882
	The Console Graphics Lite Functions .....	883
	Implementations of the Console Graphics Lite Functions .....	884
	Microsoft Compilers .....	885
	Borland Compilers.....	885
	Source Code Listings .....	885
	Listing for MSOFTCON.H .....	886
	Listing for MSOFTCON.CPP .....	886
	Listing for BORLACON.H .....	890
	Listing for BORLACON.CPP .....	891
<b>F</b>	<b>STL Algorithms and Member Functions</b>	<b>895</b>
	Algorithms .....	896
	Member Functions .....	907
	Iterators .....	909
<b>G</b>	<b>Answers to Questions and Exercises</b>	<b>913</b>
	Chapter 1.....	914
	Answers to Questions .....	914
	Chapter 2.....	914
	Answers to Questions .....	914
	Solutions to Exercises .....	916
	Chapter 3.....	917
	Answers to Questions .....	917
	Solutions to Exercises .....	918
	Chapter 4.....	921
	Answers to Questions .....	921
	Solutions to Exercises .....	922
	Chapter 5.....	924
	Answers to Questions .....	924
	Solutions to Exercises .....	925
	Chapter 6.....	928
	Answers to Questions .....	928
	Solutions to Exercises .....	929
	Chapter 7.....	932
	Answers to Questions .....	932
	Solutions to Exercises .....	933
	Chapter 8.....	937
	Answers to Questions .....	937
	Solutions to Exercises .....	938
	Chapter 9.....	943
	Answers to Questions .....	943
	Solutions to Exercises .....	944

Chapter 10.....	949
Answers to Questions .....	949
Solutions to Exercises .....	950
Chapter 11.....	954
Answers to Questions .....	954
Solutions to Exercises .....	956
Chapter 12.....	960
Answers to Questions .....	960
Solutions to Exercises .....	961
Chapter 13.....	963
Answers to Questions .....	963
Chapter 14.....	964
Answers to Questions .....	964
Solutions to Exercises .....	965
Chapter 15.....	969
Answers to Questions .....	969
Solutions to Exercises .....	970
Chapter 16.....	974
Answers to Questions .....	974

## **H Bibliography 977**

Advanced C++ .....	978
Defining Documents .....	978
The Unified Modeling Language .....	978
The History of C++ .....	979
Other Topics .....	979

## **Index 981**

# Preface

The major changes to this Fourth Edition include an earlier introduction to UML, a new section on inter-file communication in Chapter 13, and a revised approach to software development in Chapter 16.

Introducing the UML at the beginning allows the use of UML diagrams where they fit naturally with topics in the text, so there are many new UML diagrams throughout the book. The section on inter-file communication gathers together many concepts that were previously scattered throughout the book. The industry's approach to object-oriented analysis and design has evolved since the last edition, and accordingly we've modified the chapter on this topic to reflect recent developments.

C++ itself has changed very little since the last edition. However, besides the revisions just mentioned, we've made many smaller changes to clarify existing topics and correct typos and inaccuracies in the text.

# About the Author

**Robert Lafore** has been writing books about computer programming since 1982. His best-selling titles include *Assembly Language Programming for the IBM PC*, *C Programming Using Turbo C++*, *C++ Interactive Course*, and *Data Structures and Algorithms in Java*. Mr. Lafore holds degrees in mathematics and electrical engineering, and has been active in programming since the days of the PDP-5, when 4K of main memory was considered luxurious. His interests include hiking, windsurfing, and recreational mathematics.

# Dedication

*This book is dedicated to GGL and her indomitable spirit.*

## Acknowledgments to the Fourth Edition

My thanks to many readers who e-mailed comments and corrections. I am also indebted to the following professors of computer science who offered their suggestions and corrections: Bill Blomberg of Regis University in Denver; Richard Daehler-Wilking of the College of Charleston in South Carolina; Frank Hoffmann of the Royal Institute of Technology in Sweden, and David Blockus of San Jose State University in California. My special thanks to David Topham of Ohlone College in Fremont, California, for his many detailed ideas and his sharp eye for problems.

At Sams Publishing, Michael Stephens provided an expert and friendly liaison with the details of publishing. Reviewer Robin Rowe and Technical Editor Mark Cashman attempted with great care to save me from myself; any lack of success is entirely my fault. Project Manager Christina Smith made sure that everything came together in an amazingly short time, Angela Boley helped keep everything moving smoothly, and Matt Wynalda provided expert proofreading. I'm grateful to you all.

## Acknowledgments to the Third Edition

I'd like to thank the entire team at MacMillan Computer Publishing. In particular, Tracy Dunkelberger ably spearheaded the entire project and exhibited great patience with what turned out to be a lengthy schedule. Jeff Durham handled the myriad details involved in interfacing between me and the editors with skill and good humor. Andrei Kossorouko lent his expertise in C++ to ensure that I didn't make this edition worse instead of better.

## Acknowledgments to the Second Edition

My thanks to the following professors—users of this book as a text at their respective colleges and universities—for their help in planning the second edition: Dave Bridges, Frank Cioch, Jack Davidson, Terrence Fries, Jimmie Hattemer, Jack Van Luik, Kieran Mathieson, Bill McCarty, Anita Millsbaugh, Ian Moraes, Jorge Prendes, Steve Silva, and Edward Wright.

I would like to thank the many readers of the first edition who wrote in with corrections and suggestions, many of which were invaluable.

At Waite Group Press, Joanne Miller has ably ridden herd on my errant scheduling and filled in as academic liaison, and Scott Calamar, as always, has made sure that everyone knew what they were doing. Deirdre Greene provided an uncannily sharp eye as copy editor.

Thanks, too, to Mike Radtke and Harry Henderson for their expert technical reviews.

Special thanks to Edward Wright, of Western Oregon State College, for reviewing and experimenting with the new exercises.

## **Acknowledgments to the First Edition**

My primary thanks go to Mitch Waite, who poured over every inch of the manuscript with painstaking attention to detail and made a semi-infinite number of helpful suggestions.

Bill McCarty of Azusa Pacific University reviewed the content of the manuscript and its suitability for classroom use, suggested many excellent improvements, and attempted to correct my dyslexic spelling.

George Leach ran all the programs, and, to our horror, found several that didn't perform correctly in certain circumstances. I trust these problems have all been fixed; if not, the fault is entirely mine.

Scott Calamar of the Waite Group dealt with the myriad organizational aspects of writing and producing this book. His competence and unfailing good humor were an important ingredient in its completion.

I would also like to thank Nan Borreson of Borland for supplying the latest releases of the software (among other useful tidbits), Harry Henderson for reviewing the exercises, Louise Orlando of the Waite Group for ably shepherding the book through production, Merrill Peterson of Matrix Productions for coordinating the most trouble-free production run I've ever been involved with, Juan Vargas for the innovative design, and Frances Hasegawa for her uncanny ability to decipher my sketches and produce beautiful and effective art.

# Tell Us What You Think!

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

As an executive editor for Sams Publishing, I welcome your comments. You can e-mail or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

*Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.*

When you write, please be sure to include this book's title and author's name as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail: [feedback@sampublishing.com](mailto:feedback@sampublishing.com)

Mail:

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

# Introduction

This book teaches you how to write programs in the C++ programming language. However, it does more than that. In the past few years, several major innovations in software development have appeared on the scene. This book teaches C++ in the context of these new developments. Let's see what they are.

## Programming Innovations

In the old days, 20 or so years ago, programmers starting a project would sit down almost immediately and start writing code. However, as programming projects became large and more complicated, it was found that this approach did not work very well. The problem was complexity.

Large programs are probably the most complicated entities ever created by humans. Because of this complexity, programs are prone to error, and software errors can be expensive and even life threatening (in air traffic control, for example). Three major innovations in programming have been devised to cope with the problem of complexity. They are

- Object-oriented programming (OOP)
- The Unified Modeling Language (UML)
- Improved software development processes

This book teaches the C++ language with these developments in mind. You will not only learn a computer language, but new ways of conceptualizing software development.

## Object-Oriented Programming

Why has object-oriented programming become the preferred approach for most software projects? OOP offers a new and powerful way to cope with complexity. Instead of viewing a program as a series of steps to be carried out, it views it as a group of objects that have certain properties and can take certain actions. This may sound obscure until you learn more about it, but it results in programs that are clearer, more reliable, and more easily maintained.

A major goal of this book is to teach object-oriented programming. We introduce it as early as possible, and cover all its major features. The majority of our example programs are object-oriented.

## The Unified Modeling Language

The Unified Modeling Language (UML) is a graphical language consisting of many kinds of diagrams. It helps program analysts figure out what a program should do, and helps programmers design and understand how a program works. The UML is a powerful tool that can make programming easier and more effective.



We give an overview of the UML in Chapter 1, and then discuss specific features of the UML throughout the book. We introduce each UML feature where it will help to clarify the OOP topic being discussed. In this way you learn the UML painlessly at the same time the UML helps you to learn C++.

## Languages and Development Platforms

Of the object-oriented programming languages, C++ is by far the most widely used. Java, a recent addition to the field of OO languages, lacks certain features—such as pointers, templates, and multiple inheritance—that make it less powerful and versatile than C++. (If you ever do want to learn Java, its syntax is very similar to that of C++, so learning C++ gives you a head start in Java.)

Several other OO languages have been introduced recently, such as C#, but they have not yet attained the wide acceptance of C++.

Until recently the standards for C++ were in a constant state of evolution. This meant that each compiler vendor handled certain details differently. However, in November 1997, the ANSI/ISO C++ standards committee approved the final draft of what is now known as Standard C++. (ANSI stands for American National Standards Institute, and ISO stands for International Standards Institute.) Standard C++ adds many new features to the language, such as the Standard Template Library (STL). In this book we follow Standard C++ (in all but a few places, which we'll note as we go along).

The most popular development environments for C++ are manufactured by Microsoft and Borland (Inprise) and run on the various flavors of Microsoft Windows. In this book we've attempted to ensure that all sample programs run on the current versions of both Borland and Microsoft compilers. (See Appendix C, "Microsoft Visual C++," and Appendix D, "Borland C++Builder," for more on these compilers.)

## What This Book Does

This book teaches object-oriented programming with the C++ programming language, using either Microsoft or Borland compilers. It also introduces the UML and software development processes. It is suitable for professional programmers, students, and kitchen-table enthusiasts.

## New Concepts

OOP involves concepts that are new to programmers of traditional languages such as Pascal, Basic, and C. These ideas, such as classes, inheritance, and polymorphism, lie at the heart of object-oriented programming. But it's easy to lose sight of these concepts when discussing the specifics of an object-oriented language. Many books overwhelm the reader with the details of language features, while ignoring the reason these features exist. This book attempts to keep an eye on the big picture and relate the details to the larger concepts.

## The Gradual Approach

We take a gradual approach in this book, starting with very simple programming examples and working up to full-fledged object-oriented applications. We introduce new concepts slowly so that you will have time to digest one idea before going on to the next. We use illustrations whenever possible to help clarify new ideas. There are questions and programming exercises at the end of most chapters to enhance the book's usefulness in the classroom. Answers to the questions and to the first few (starred) exercises can be found in Appendix G. The exercises vary in difficulty to pose a variety of challenges for the student.

## What You Need to Know to Use This Book

You can use this book even if you have no previous programming experience. However, such experience, in Visual Basic for example, certainly won't hurt.

You do not need to know the C language to use this book. Many books on C++ assume that you already know C, but this one does not. It teaches C++ from the ground up. If you do know C, it won't hurt, but you may be surprised at how little overlap there is between C and C++.

You should be familiar with the basic operations of Microsoft Windows, such as starting applications and copying files.

## Software and Hardware

You will need a C++ compiler. The programs in this book have been tested with Microsoft Visual C++ and Borland C++Builder. Both compilers come in low-priced "Learning Editions" suitable for students.

Appendix C provides detailed information on operating the Microsoft compiler, while Appendix D does the same for the Inprise (Borland) product. Other compilers, if they adhere to Standard C++, will probably handle most of the programs in this book as written.

Your computer should have enough processor speed, memory, and hard disk space to run the compiler you've chosen. You can check the manufacturer's specifications to determine these requirements.

## Console-Mode Programs

There are numerous example programs throughout the book. They are console-mode programs, which run in a character-mode window within the compiler environment, or directly within an MS-DOS box. This avoids the complexity of full-scale graphics-oriented Windows programs.

## Example Program Source Code

You can obtain the source code for the example programs from the Sams Publishing Web site at

<http://www.sampublishing.com>

Type the ISBN (found at the front of the book) or the book's title and click Search to find the data on this book. Then click Source Code to download the program examples.

## Console Graphics Lite

A few example programs draw pictures using a graphics library we call Console Graphics Lite. The graphics rely on console characters, so they are not very sophisticated, but they allow some interesting programs. The files for this library are provided on the publisher's Web site, along with the source files for the example programs.

To compile and run these graphics examples, you'll need to include a header file in your program, either `MSOFTCON.H` or `BORLACON.H`, depending on your compiler. You'll also need to add either `MSOFTCON.CPP` or `BORLACON.CPP` to the project for the graphics example. Appendix E, "Console Graphics Lite," provides listings of these files and tells how to use them. Appendixes C and D explain how to work with files and projects in a specific compiler's environment.

## Programming Exercises

Each chapter contains roughly 12 exercises, each requiring the creation of a complete C++ program. Solutions for the first three or four exercises in each chapter are provided in Appendix G. For the remainder of the exercises, readers are on their own. (However, if you are teaching a C++ course, see the "Note to Teachers" at the end of this Introduction.)

## Easier Than You Think

You may have heard that C++ is difficult to learn, but it's really quite similar to other languages, with two or three "grand ideas" thrown in. These new ideas are fascinating in themselves, and we think you'll have fun learning about them. They are also becoming part of the programming culture; they're something everyone should know a little bit about, like evolution and psychoanalysis. We hope this book will help you enjoy learning about these new ideas, at the same time that it teaches you the details of programming in C++.

## A Note to Teachers

Teachers, and others who already know something about C++ or C, may be interested in some details of the approach we use in this book and how it's organized.

### Standard C++

All the programs in this book are compatible with Standard C++, with a few minor exceptions that are needed to accommodate compiler quirks. We devote a chapter to the STL (Standard Template Library), which is included in Standard C++.

### The Unified Modeling Language (UML)

In the previous edition, we introduced the UML in the final chapter. In this edition we have integrated the UML into the body of the book, introducing UML topics in appropriate places. For example, UML class diagrams are introduced where we first show different classes communicating, and generalization is covered in the chapter on inheritance.

Chapter 1, “The Big Picture,” includes a list showing where the various UML topics are introduced.

### Software Development Processes

Formal software development processes are becoming an increasingly important aspect of programming. Also, students are frequently mystified by the process of designing an object-oriented program. For these reasons we include a chapter on software development processes, with an emphasis on object-oriented programming. In the last edition we focused on CRC cards, but the emphasis in software development has shifted more in the direction of use case analysis, so we use that to analyze our programming projects.

### C++ Is Not the Same as C

A few institutions still want their students to learn C before learning C++. In our view this is a mistake. C and C++ are entirely separate languages. It's true that their syntax is similar, and C is actually a subset of C++. But the similarity is largely a historical accident. In fact, the basic approach in a C++ program is radically different from that in a C program.

C++ has overtaken C as the preferred language for serious software development. Thus we don't believe it is necessary or advantageous to teach C before teaching C++. Students who don't know C are saved the time and trouble of learning C and then learning C++, an inefficient approach. Students who already know C may be able to skim parts of some chapters, but they will find that a remarkable percentage of the material is new.

## Optimize Organization for OOP

We could have begun the book by teaching the procedural concepts common to C and C++, and moved on to the new OOP concepts once the procedural approach had been digested. That seemed counterproductive, however, because one of our goals is to begin true object-oriented programming as quickly as possible. Accordingly, we provide a minimum of procedural groundwork before getting to classes in Chapter 6. Even the initial chapters are heavily steeped in C++, as opposed to C, usage.

We introduce some concepts earlier than is traditional in books on C. For example, structures are a key feature for understanding C++ because classes are syntactically an extension of structures. For this reason, we introduce structures in Chapter 5 so that they will be familiar when we discuss classes.

Some concepts, such as pointers, are introduced later than in traditional C books. It's not necessary to understand pointers to follow the essentials of OOP, and pointers are usually a stumbling block for C and C++ students. Therefore, we defer a discussion of pointers until the main concepts of OOP have been thoroughly digested.

## Substitute Superior C++ Features

Some features of C have been superseded by new approaches in C++. For instance, the `printf()` and `scanf()` functions, input/output workhorses in C, are seldom used in C++ because `cout` and `cin` do a better job. Consequently, we leave out descriptions of these functions. Similarly, `#define` constants and macros in C have been largely superseded by the `const` qualifier and inline functions in C++, and need be mentioned only briefly.

## Minimize Irrelevant Capabilities

Because the focus in this book is on object-oriented programming, we can leave out some features of C that are seldom used and are not particularly relevant to OOP. For instance, it isn't necessary to understand the C bit-wise operators (used to operate on individual bits) to learn object-oriented programming. These and a few other features can be dropped from our discussion, or mentioned only briefly, with no loss in understanding of the major features of C++.

The result is a book that focuses on the fundamentals of OOP, moving the reader gently but briskly toward an understanding of new concepts and their application to real programming problems.

## Programming Exercises

No answers to the unstarred exercises are provided in this book. However, qualified instructors can obtain suggested solutions from the Sams Publishing Web site. Type the ISBN or title and click Search to move to this book's page, then click Downloads.

The exercises vary considerably in their degree of difficulty. In each chapter the early exercises are fairly easy, while later ones are more challenging. Instructors will probably want to assign only those exercises suited to the level of a particular class.



# The Big Picture

## CHAPTER

# 1

## IN THIS CHAPTER

- **Why Do We Need Object-Oriented Programming?** 10
- **Characteristics of Object-Oriented Languages** 16
- **C++ and C** 22
- **Laying the Groundwork** 23
- **The Unified Modeling Language (UML)** 23



This book teaches you how to program in C++, a computer language that supports *object-oriented programming* (OOP). Why do we need OOP? What does it do that traditional languages such as C, Pascal, and BASIC don't? What are the principles behind OOP? Two key concepts in OOP are *objects* and *classes*. What do these terms mean? What is the relationship between C++ and the older C language?

This chapter explores these questions and provides an overview of the features to be discussed in the balance of the book. What we say here will necessarily be rather general (although mercifully brief). If you find the discussion somewhat abstract, don't worry. The concepts we mention here will come into focus as we demonstrate them in detail in subsequent chapters.

## Why Do We Need Object-Oriented Programming?

Object-oriented programming was developed because limitations were discovered in earlier approaches to programming. To appreciate what OOP does, we need to understand what these limitations are and how they arose from traditional programming languages.

### Procedural Languages

C, Pascal, FORTRAN, and similar languages are *procedural languages*. That is, each statement in the language tells the computer to do something: Get some input, add these numbers, divide by six, display that output. A program in a procedural language is a list of instructions.

For very small programs, no other organizing principle (often called a *paradigm*) is needed. The programmer creates the list of instructions, and the computer carries them out.

### Division into Functions

When programs become larger, a single list of instructions becomes unwieldy. Few programmers can comprehend a program of more than a few hundred statements unless it is broken down into smaller units. For this reason the *function* was adopted as a way to make programs more comprehensible to their human creators. (The term function is used in C++ and C. In other languages the same concept may be referred to as a subroutine, a subprogram, or a procedure.) A procedural program is divided into functions, and (ideally, at least) each function has a clearly defined purpose and a clearly defined interface to the other functions in the program.

The idea of breaking a program into functions can be further extended by grouping a number of functions together into a larger entity called a *module* (which is often a file), but the principle is similar: a grouping of components that execute lists of instructions.

Dividing a program into functions and modules is one of the cornerstones of *structured programming*, the somewhat loosely defined discipline that influenced programming organization for several decades before the advent of object-oriented programming.

## Problems with Structured Programming

As programs grow ever larger and more complex, even the structured programming approach begins to show signs of strain. You may have heard about, or been involved in, horror stories of program development. The project is too complex, the schedule slips, more programmers are added, complexity increases, costs skyrocket, the schedule slips further, and disaster ensues. (See *The Mythical Man-Month* by Frederick P. Brooks, Jr. [Addison Wesley, 1982] for a vivid description of this process.)

Analyzing the reasons for these failures reveals that there are weaknesses in the procedural paradigm itself. No matter how well the structured programming approach is implemented, large programs become excessively complex.

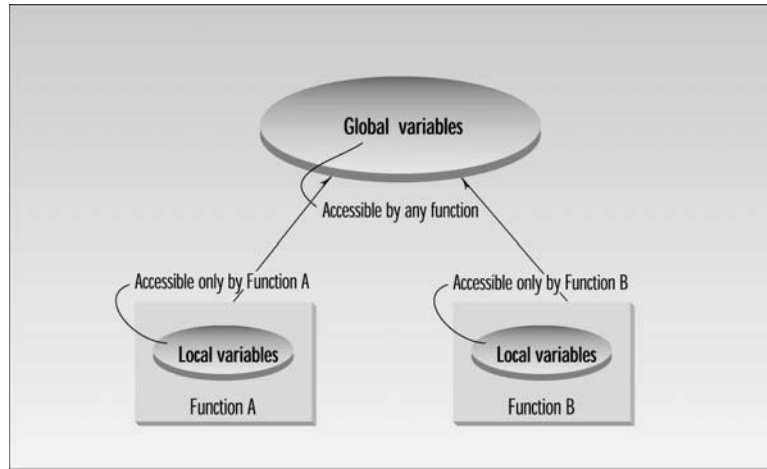
What are the reasons for these problems with procedural languages? There are two related problems. First, functions have unrestricted access to global data. Second, unrelated functions and data, the basis of the procedural paradigm, provide a poor model of the real world.

Let's examine these problems in the context of an inventory program. One important global data item in such a program is the collection of items in the inventory. Various functions access this data to input a new item, display an item, modify an item, and so on.

## Unrestricted Access

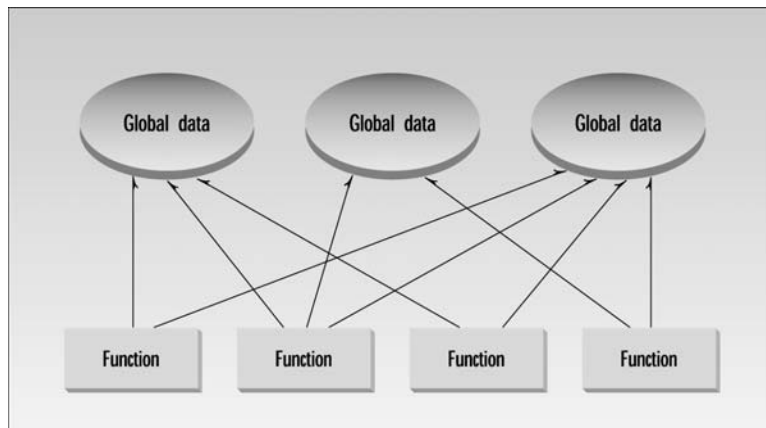
In a procedural program, one written in C for example, there are two kinds of data. *Local data* is hidden inside a function, and is used exclusively by the function. In the inventory program a display function might use local data to remember which item it was displaying. Local data is closely related to its function and is safe from modification by other functions.

However, when two or more functions must access the same data—and this is true of the most important data in a program—then the data must be made *global*, as our collection of inventory items is. Global data can be accessed by *any* function in the program. (We ignore the issue of grouping functions into modules, which doesn't materially affect our argument.) The arrangement of local and global variables in a procedural program is shown in Figure 1.1.

**FIGURE 1.1**

*Global and local variables.*

In a large program, there are many functions and many global data items. The problem with the procedural paradigm is that this leads to an even larger number of potential connections between functions and data, as shown in Figure 1.2.

**FIGURE 1.2**

*The procedural paradigm.*

This large number of connections causes problems in several ways. First, it makes a program's structure difficult to conceptualize. Second, it makes the program difficult to modify. A change made in a global data item may necessitate rewriting all the functions that access that item.

For example, in our inventory program, someone may decide that the product codes for the inventory items should be changed from 5 digits to 12 digits. This may necessitate a change from a short to a long data type.

Now all the functions that operate on the data must be modified to deal with a long instead of a short. It's similar to what happens when your local supermarket moves the bread from aisle 4 to aisle 7. Everyone who patronizes the supermarket must then figure out where the bread has gone, and adjust their shopping habits accordingly.

When data items are modified in a large program it may not be easy to tell which functions access the data, and even when you figure this out, modifications to the functions may cause them to work incorrectly with other global data items. Everything is related to everything else, so a modification anywhere has far-reaching, and often unintended, consequences.

## Real-World Modeling

The second—and more important—problem with the procedural paradigm is that its arrangement of separate data and functions does a poor job of modeling things in the real world. In the physical world we deal with objects such as people and cars. Such objects aren't like data and they aren't like functions. Complex real-world objects have both *attributes* and *behavior*.

### Attributes

Examples of attributes (sometimes called *characteristics*) are, for people, eye color and job title; and, for cars, horsepower and number of doors. As it turns out, attributes in the real world are equivalent to data in a program: they have a certain specific values, such as blue (for eye color) or four (for the number of doors).

### Behavior

Behavior is something a real-world object does in response to some stimulus. If you ask your boss for a raise, she will generally say yes or no. If you apply the brakes in a car, it will generally stop. Saying something and stopping are examples of behavior. Behavior is like a function: you call a function to do something (display the inventory, for example) and it does it.

So neither data nor functions, by themselves, model real-world objects effectively.

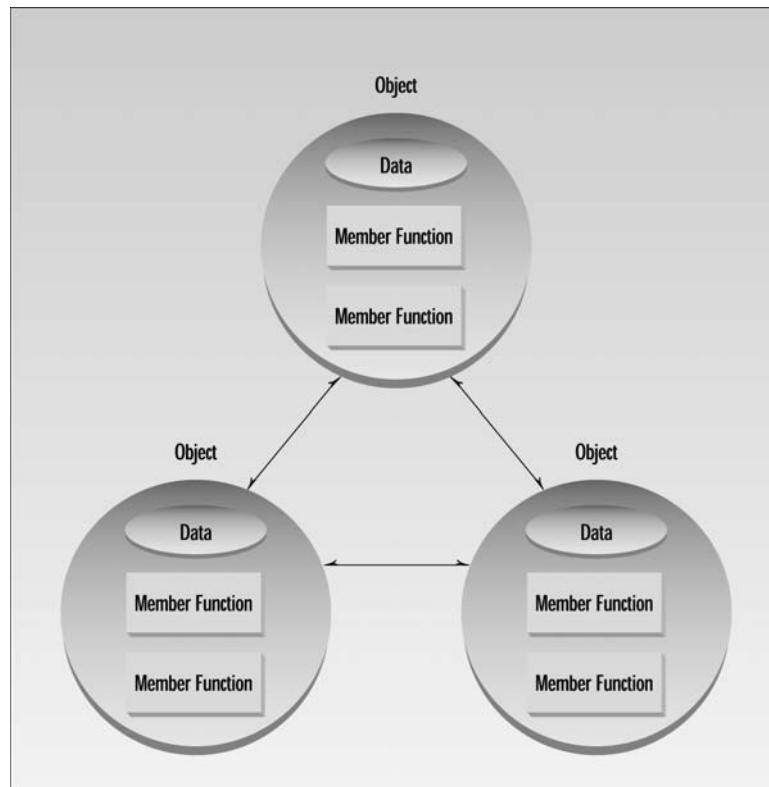
## The Object-Oriented Approach

The fundamental idea behind object-oriented languages is to combine into a single unit both *data* and the *functions that operate on that data*. Such a unit is called an *object*.

An object's functions, called *member functions* in C++, typically provide the only way to access its data. If you want to read a data item in an object, you call a member function in the object. It will access the data and return the value to you. You can't access the data directly. The data is *hidden*, so it is safe from accidental alteration. Data and its functions are said to be *encapsulated* into a single entity. *Data encapsulation* and *data hiding* are key terms in the description of object-oriented languages.

If you want to modify the data in an object, you know exactly what functions interact with it: the member functions in the object. No other functions can access the data. This simplifies writing, debugging, and maintaining the program.

A C++ program typically consists of a number of objects, which communicate with each other by calling one another's member functions. The organization of a C++ program is shown in Figure 1.3.



**FIGURE 1.3**

*The object-oriented paradigm.*

We should mention that what are called member functions in C++ are called *methods* in some other object-oriented (OO) languages (such as Smalltalk, one of the first OO languages). Also, data items are referred to as *attributes* or *instance variables*. Calling an object's member function is referred to as *sending a message* to the object. These terms are not official C++ terminology, but they are used with increasing frequency, especially in object-oriented design.

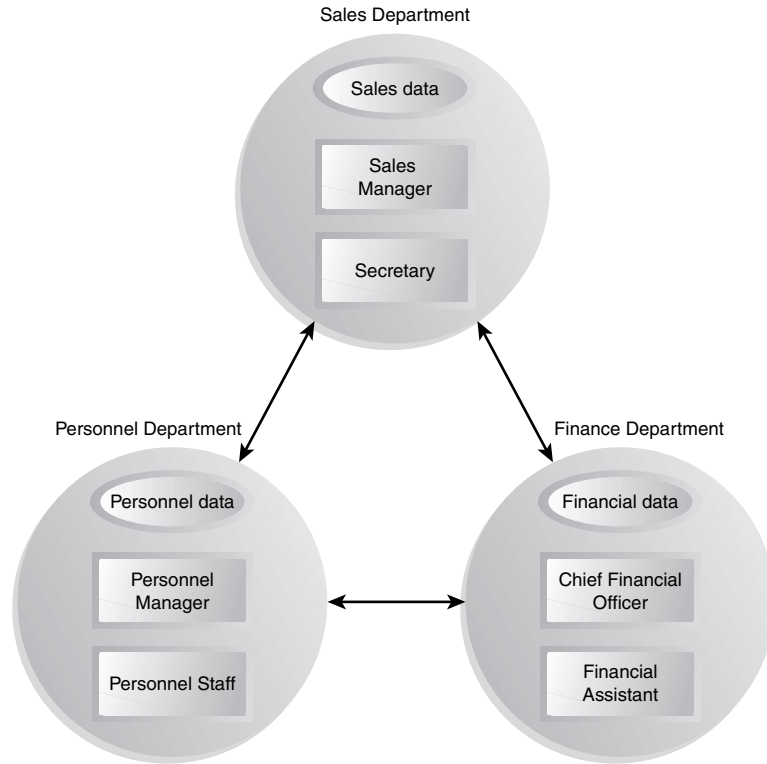
## An Analogy

You might want to think of objects as departments—such as sales, accounting, personnel, and so on—in a company. Departments provide an important approach to corporate organization. In most companies (except very small ones), people don't work on personnel problems one day, the payroll the next, and then go out in the field as salespeople the week after. Each department has its own personnel, with clearly assigned duties. It also has its own data: the accounting department has payroll figures, the sales department has sales figures, the personnel department keeps records of each employee, and so on.

The people in each department control and operate on that department's data. Dividing the company into departments makes it easier to comprehend and control the company's activities, and helps maintain the integrity of the information used by the company. The accounting department, for instance, is responsible for the payroll data. If you're a sales manager, and you need to know the total of all the salaries paid in the southern region in July, you don't just walk into the accounting department and start rummaging through file cabinets. You send a memo to the appropriate person in the department, then wait for that person to access the data and send you a reply with the information you want. This ensures that the data is accessed accurately and that it is not corrupted by inept outsiders. This view of corporate organization is shown in Figure 1.4. In the same way, objects provide an approach to program organization while helping to maintain the integrity of the program's data.

## OOP: An Approach to Organization

Keep in mind that object-oriented programming is not primarily concerned with the details of program operation. Instead, it deals with the overall organization of the program. Most individual program statements in C++ are similar to statements in procedural languages, and many are identical to statements in C. Indeed, an entire member function in a C++ program may be very similar to a procedural function in C. It is only when you look at the larger context that you can determine whether a statement or a function is part of a procedural C program or an object-oriented C++ program.

**FIGURE 1.4**

*The corporate paradigm.*

## Characteristics of Object-Oriented Languages

Let's briefly examine a few of the major elements of object-oriented languages in general, and C++ in particular.

### Objects

When you approach a programming problem in an object-oriented language, you no longer ask how the problem will be divided into functions, but how it will be divided into objects. Thinking in terms of objects, rather than functions, has a surprisingly helpful effect on how easily programs can be designed. This results from the close match between objects in the programming sense and objects in the real world. This process is described in detail in Chapter 16, "Object-Oriented Software Development."

What kinds of things become objects in object-oriented programs? The answer to this is limited only by your imagination, but here are some typical categories to start you thinking:

- **Physical objects**

- Automobiles in a traffic-flow simulation

- Electrical components in a circuit-design program

- Countries in an economics model

- Aircraft in an air traffic control system

- **Elements of the computer-user environment**

- Windows

- Menus

- Graphics objects (lines, rectangles, circles)

- The mouse, keyboard, disk drives, printer

- **Data-storage constructs**

- Customized arrays

- Stacks

- Linked lists

- Binary trees

- **Human entities**

- Employees

- Students

- Customers

- Salespeople

- **Collections of data**

- An inventory

- A personnel file

- A dictionary

- A table of the latitudes and longitudes of world cities

- **User-defined data types**

- Time

- Angles

- Complex numbers

- Points on the plane



- **Components in computer games**

- Cars in an auto race

- Positions in a board game (chess, checkers)

- Animals in an ecological simulation

- Opponents and friends in adventure games

The match between programming objects and real-world objects is the happy result of combining data and functions: The resulting objects offer a revolution in program design. No such close match between programming constructs and the items being modeled exists in a procedural language.

## Classes

In OOP we say that objects are members of *classes*. What does this mean? Let's look at an analogy. Almost all computer languages have built-in data types. For instance, a data type `int`, meaning integer, is predefined in C++ (as we'll see in Chapter 3, "Loops and Decisions"). You can declare as many variables of type `int` as you need in your program:

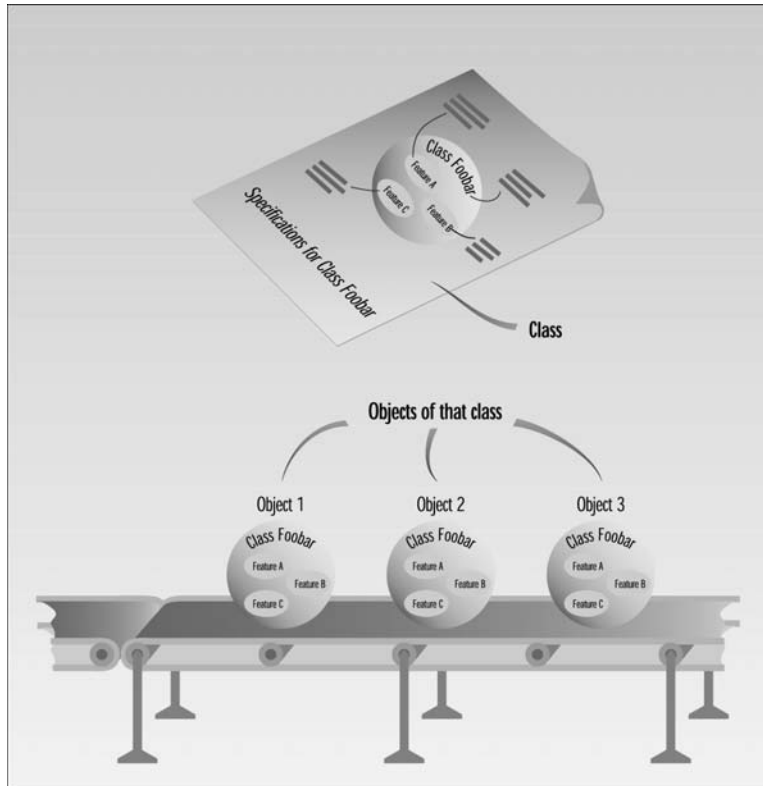
```
int day;  
int count;  
int divisor;  
int answer;
```

In a similar way, you can define many objects of the same class, as shown in Figure 1.5. A class serves as a plan, or blueprint. It specifies what data and what functions will be included in objects of that class. Defining the class doesn't create any objects, just as the mere existence of data type `int` doesn't create any variables.

A class is thus a description of a number of similar objects. This fits our non-technical understanding of the word *class*. Prince, Sting, and Madonna are members of the rock musician class. There is no one person called "rock musician," but specific people with specific names are members of this class if they possess certain characteristics. An object is often called an "instance" of a class.

## Inheritance

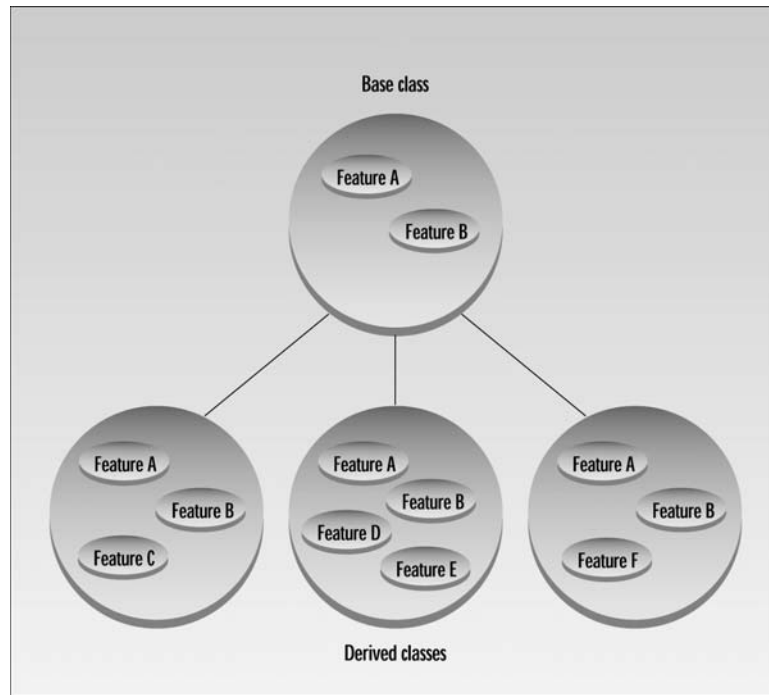
The idea of classes leads to the idea of *inheritance*. In our daily lives, we use the concept of classes divided into subclasses. We know that the animal class is divided into mammals, amphibians, insects, birds, and so on. The vehicle class is divided into cars, trucks, buses, motorcycles, and so on.

**FIGURE 1.5**

*A class and its objects.*

The principle in this sort of division is that each subclass shares common characteristics with the class from which it's derived. Cars, trucks, buses, and motorcycles all have wheels and a motor; these are the defining characteristics of vehicles. In addition to the characteristics shared with other members of the class, each subclass also has its own particular characteristics: Buses, for instance, have seats for many people, while trucks have space for hauling heavy loads.

This idea is shown in Figure 1.6. Notice in the figure that features A and B, which are part of the base class, are common to all the derived classes, but that each derived class also has features of its own.

**FIGURE 1.6***Inheritance.*

In a similar way, an OOP class can become a parent of several subclasses. In C++ the original class is called the *base class*; other classes can be defined that share its characteristics, but add their own as well. These are called *derived classes*.

Don't confuse the relation of objects to classes, on the one hand, with the relation of a base class to derived classes, on the other. Objects, which exist in the computer's memory, each embody the exact characteristics of their class, which serves as a template. Derived classes inherit some characteristics from their base class, but add new ones of their own.

Inheritance is somewhat analogous to using functions to simplify a traditional procedural program. If we find that three different sections of a procedural program do almost exactly the same thing, we recognize an opportunity to extract the common elements of these three sections and put them into a single function. The three sections of the program can call the function to execute the common actions, and they can perform their own individual processing as well. Similarly, a base class contains elements common to a group of derived classes. As functions do in a procedural program, inheritance shortens an object-oriented program and clarifies the relationship among program elements.

## Reusability

Once a class has been written, created, and debugged, it can be distributed to other programmers for use in their own programs. This is called *reusability*. It is similar to the way a library of functions in a procedural language can be incorporated into different programs.

However, in OOP, the concept of inheritance provides an important extension to the idea of reusability. A programmer can take an existing class and, without modifying it, add additional features and capabilities to it. This is done by deriving a new class from the existing one. The new class will inherit the capabilities of the old one, but is free to add new features of its own.

For example, you might have written (or purchased from someone else) a class that creates a menu system, such as that used in Windows or other Graphic User Interfaces (GUIs). This class works fine, and you don't want to change it, but you want to add the capability to make some menu entries flash on and off. To do this, you simply create a new class that inherits all the capabilities of the existing one but adds flashing menu entries.

The ease with which existing software can be reused is an important benefit of OOP. Many companies find that being able to reuse classes on a second project provides an increased return on their original programming investment. We'll have more to say about this in later chapters.

## Creating New Data Types

One of the benefits of objects is that they give the programmer a convenient way to construct new data types. Suppose you work with two-dimensional positions (such as *x* and *y* coordinates, or latitude and longitude) in your program. You would like to express operations on these positional values with normal arithmetic operations, such as

```
position1 = position2 + origin
```

where the variables *position1*, *position2*, and *origin* each represent a pair of independent numerical quantities. By creating a class that incorporates these two values, and declaring *position1*, *position2*, and *origin* to be objects of this class, we can, in effect, create a new data type. Many features of C++ are intended to facilitate the creation of new data types in this manner.

## Polymorphism and Overloading

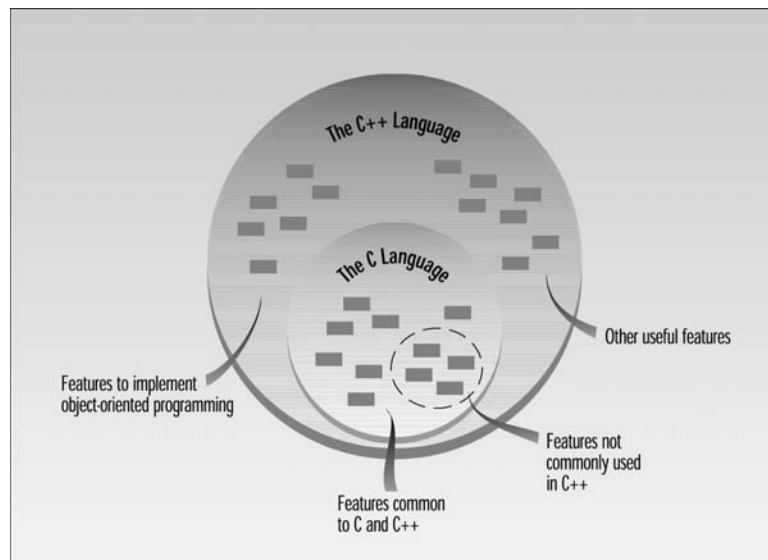
Note that the *=* (equal) and *+* (plus) operators, used in the position arithmetic shown above, don't act the same way they do in operations on built-in types such as *int*. The objects *position1* and so on are not predefined in C++, but are programmer-defined

objects of class `Position`. How do the `=` and `+` operators know how to operate on objects? The answer is that we can define new behaviors for these operators. These operations will be member functions of the `Position` class.

Using operators or functions in different ways, depending on what they are operating on, is called *polymorphism* (one thing with several distinct forms). When an existing operator, such as `+` or `=`, is given the capability to operate on a new data type, it is said to be *overloaded*. Overloading is a kind of polymorphism; it is also an important feature of OOP.

## C++ and C

C++ is derived from the C language. Strictly speaking, it is a superset of C: Almost every correct statement in C is also a correct statement in C++, although the reverse is not true. The most important elements added to C to create C++ concern classes, objects, and object-oriented programming. (C++ was originally called “C with classes.”) However, C++ has many other new features as well, including an improved approach to input/output (I/O) and a new way to write comments. Figure 1.7 shows the relationship of C and C++.



**FIGURE 1.7**

*The relationship between C and C++.*

In fact, the practical differences between C and C++ are larger than you might think. Although you can write a program in C++ that looks like a program in C, hardly anyone does. C++ programmers not only make use of the new features of C++, they also emphasize the traditional C features in different proportions than do C programmers.

If you already know C, you will have a head start in learning C++ (although you may also have some bad habits to unlearn), but much of the material will be new.

## Laying the Groundwork

Our goal is to help you begin writing OOP programs as soon as possible. However, as we noted, much of C++ is inherited from C, so while the overall structure of a C++ program may be OOP, down in the trenches you need to know some old-fashioned procedural fundamentals. Chapters 2–5 therefore deal with the “traditional” aspects of C++, many of which are also found in C. You will learn about variables and I/O, about control structures such as loops and decisions, and about functions themselves. You will also learn about structures, since the same syntax that’s used for structures is used for classes.

If you already know C, you might be tempted to skip these chapters. However, you will find that there are many differences, some obvious and some rather subtle, between C and C++. Our advice is to read these chapters, skimming what you know, and concentrating on the ways C++ differs from C.

The specific discussion of OOP starts in Chapter 6, “Objects and Classes.” From then on the examples will be object oriented.

## The Unified Modeling Language (UML)

The UML is a graphical “language” for modeling computer programs. “Modeling” means to create a simplified representation of something, as a blueprint models a house. The UML provides a way to visualize the higher-level organization of programs without getting mired down in the details of actual code.

The UML began as three separate modeling languages, one created by Grady Booch at Rational Software, one by James Rumbaugh at General Electric, and one by Ivar Jacobson at Ericson. Eventually Rumbaugh and Jacobson joined Booch at Rational, where they became known as the three amigos. During the late 1990s they unified (hence the name) their modeling languages into the Unified Modeling Language. The result was adopted by the Object Management Group (OMG), a consortium of companies devoted to industry standards.

Why do we need the UML? One reason is that in a large computer program it's often hard to understand, simply by looking at the code, how the parts of the program relate to each other. As we've seen, object-oriented programming is a vast improvement over procedural programs. Nevertheless, figuring out what a program is supposed to do requires, at best, considerable study of the program listings.

The trouble with code is that it's very detailed. It would be nice if there were a way to see a bigger picture, one that depicts the major parts of the program and how they work together. The UML answers this need.

The most important part of the UML is a set of different kinds of diagrams. Class diagrams show the relationships among classes, object diagrams show how specific objects relate, sequence diagrams show the communication among objects over time, use case diagrams show how a program's users interact with the program, and so on. These diagrams provide a variety of ways to look at a program and its operation.

The UML plays many roles besides helping us to understand how a program works. As we'll see in Chapter 16, it can help in the initial design of a program. In fact, the UML is useful throughout all phases of software development, from initial specification to documentation, testing, and maintenance.

The UML is not a software development process. Many such processes exist for specifying the stages of the development process. The UML is simply a way to look at the software being developed. Although it can be applied to any kind of programming language, the UML is especially attuned to OOP.

As we noted in the Introduction, we introduce specific features of the UML in stages throughout the book.

- Chapter 1: (this section) introduction to the UML
- Chapter 8: class diagrams, associations, and navigability
- Chapter 9: generalization, aggregation, and composition
- Chapter 10: state diagrams and multiplicity
- Chapter 11: object diagrams
- Chapter 13: more complex state diagrams
- Chapter 14: templates, dependencies, and stereotypes
- Chapter 16: use cases, use case diagrams, activity diagrams, and sequence diagrams

## Summary

OOP is a way of organizing programs. The emphasis is on the way programs are designed, not on coding details. In particular, OOP programs are organized around objects, which contain both data and functions that act on that data. A class is a template for a number of objects.

Inheritance allows a class to be derived from an existing class without modifying it. The derived class has all the data and functions of the parent class, but adds new ones of its own. Inheritance makes possible reusability, or using a class over and over in different programs.

C++ is a superset of C. It adds to the C language the capability to implement OOP. It also adds a variety of other features. In addition, the emphasis is changed in C++ so that some features common to C, although still available in C++, are seldom used, while others are used far more frequently. The result is a surprisingly different language.

The Unified Modeling Language (UML) is a standardized way to visualize a program's structure and operation using diagrams.

The general concepts discussed in this chapter will become more concrete as you learn more about the details of C++. You may want to refer back to this chapter as you progress further into this book.

## Questions

Answers to these questions can be found in Appendix G. Note that throughout this book, multiple-choice questions can have more than one correct answer.

1. Pascal, BASIC, and C are p\_\_\_\_\_ languages, while C++ is an o\_\_\_\_\_ language.
2. A widget is to the blueprint for a widget as an object is to
  - a. a member function.
  - b. a class.
  - c. an operator.
  - d. a data item.
3. The two major components of an object are \_\_\_\_\_ and functions that \_\_\_\_\_.
4. In C++, a function contained within a class is called
  - a. a member function.
  - b. an operator.
  - c. a class function.
  - d. a method.



5. Protecting data from access by unauthorized functions is called \_\_\_\_.
6. Which of the following are good reasons to use an object-oriented language?
  - a. You can define your own data types.
  - b. Program statements are simpler than in procedural languages.
  - c. An OO program can be taught to correct its own errors.
  - d. It's easier to conceptualize an OO program.
7. \_\_\_\_ model entities in the real world more closely than do functions.
8. True or false: A C++ program is similar to a C program except for the details of coding.
9. Bundling data and functions together is called \_\_\_\_.
10. When a language has the capability to produce new data types, it is said to be
  - a. reprehensible.
  - b. encapsulated.
  - c. overloaded.
  - d. extensible.
11. True or false: You can easily tell, from any two lines of code, whether a program is written in C or C++.
12. The ability of a function or operator to act in different ways on different data types is called \_\_\_\_\_.
13. A normal C++ operator that acts in special ways on newly defined data types is said to be
  - a. glorified.
  - b. encapsulated.
  - c. classified.
  - d. overloaded.
14. Memorizing the new terms used in C++ is
  - a. critically important.
  - b. something you can return to later.
  - c. the key to wealth and success.
  - d. completely irrelevant.

15. The Unified Modeling Language is
- a. a program that builds physical models.
  - b. a way to look at the organization of a program.
  - c. the combination of C++ and FORTRAN.
  - d. helpful in developing software systems.



### IN THIS CHAPTER

- **Getting Started** 30
- **Basic Program Construction** 30
- **Output Using `cout`** 33
- **Directives** 35
- **Comments** 36
- **Integer Variables** 38
- **Character Variables** 42
- **Input with `cin`** 45
- **Floating Point Types** 48
- **Type `bool`** 51
- **The `setw` Manipulator** 52
- **Variable Type Summary** 54
- **Type Conversion** 56
- **Arithmetic Operators** 60
- **Library Functions** 65

In any language there are some fundamentals you need to know before you can write even the most elementary programs. This chapter introduces three such fundamentals: basic program construction, variables, and input/output (I/O). It also touches on a variety of other language features, including comments, arithmetic operators, the increment operator, data conversion, and library functions.

These topics are not conceptually difficult, but you may find that the style in C++ is a little austere compared with, say, BASIC or Pascal. Before you learn what it's all about, a C++ program may remind you more of a mathematics formula than a computer program. Don't worry about this. You'll find that as you gain familiarity with C++, it starts to look less forbidding, while other languages begin to seem unnecessarily fancy and verbose.

## Getting Started

As we noted in the Introduction, you can use either a Microsoft or a Borland compiler with this book. Appendixes C and D provide details about their operation. (Other compilers may work as well.) Compilers take source code and transform it into executable files, which your computer can run as it does other programs. Source files are text files (extension .CPP) that correspond with the listings printed in this book. Executable files have the .EXE extension, and can be executed either from within your compiler, or, if you're familiar with MS-DOS, directly from a DOS window.

The programs run without modification on the Microsoft compiler or in an MS-DOS window. If you're using the Borland compiler, you'll need to modify the programs slightly before running them; otherwise the output won't remain on the screen long enough to see. Make sure to read Appendix D, "Borland C++Builder," to see how this is done.

## Basic Program Construction

Let's look at a very simple C++ program. This program is called FIRST, so its source file is FIRST.CPP. It simply prints a sentence on the screen. Here it is:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Every age has a language of its own\n";
    return 0;
}
```

Despite its small size, this program demonstrates a great deal about the construction of C++ programs. Let's examine it in detail.

## Functions

Functions are one of the fundamental building blocks of C++. The FIRST program consists almost entirely of a single function called `main()`. The only parts of this program that are not part of the function are the first two lines—the ones that start with `#include` and `using`. (We'll see what these lines do in a moment.)

We noted in Chapter 1, “The Big Picture,” that a function can be part of a class, in which case it is called a *member function*. However, functions can also exist independently of classes. We are not yet ready to talk about classes, so we will show functions that are separate standalone entities, as `main()` is here.

### Function Name

The parentheses following the word `main` are the distinguishing feature of a function. Without the parentheses the compiler would think that `main` refers to a variable or to some other program element. When we discuss functions in the text, we'll follow the same convention that C++ uses: We'll put parentheses following the function name. Later on we'll see that the parentheses aren't always empty. They're used to hold function *arguments*: values passed from the calling program to the function.

The word `int` preceding the function name indicates that this particular function has a return value of type `int`. Don't worry about this now; we'll learn about data types later in this chapter and return values in Chapter 5, “Functions.”

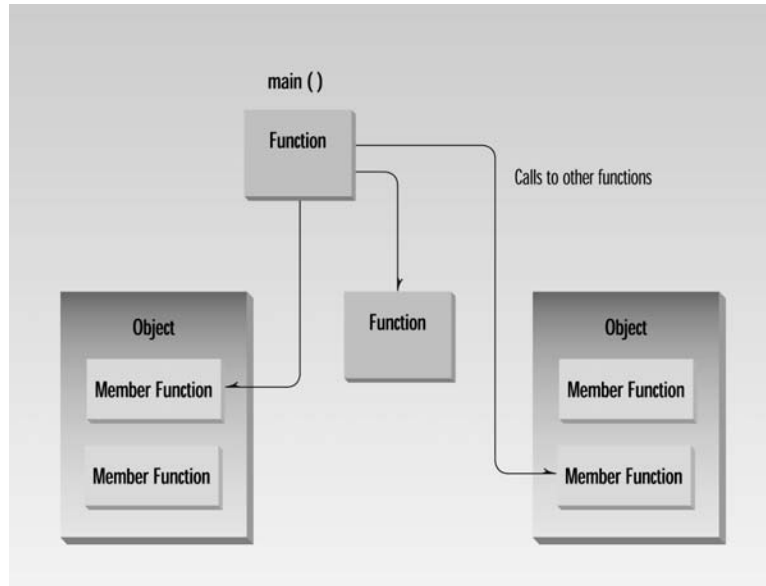
### Braces and the Function Body

The *body* of a function is surrounded by *braces* (sometimes called *curly brackets*). These braces play the same role as the `BEGIN` and `END` keywords in some other languages: They surround or *delimit* a block of program statements. Every function must use this pair of braces around the function body. In this example there are only two statements in the function body: the line starting with `cout`, and the line starting with `return`. However, a function body can consist of many statements.

### Always Start with `main()`

When you run a C++ program, the first statement executed will be at the beginning of a function called `main()`. (At least that's true of the console mode programs in this book.) The program may consist of many functions, classes, and other program elements, but on startup, control always goes to `main()`. If there is no function called `main()` in your program, an error will be reported when you run the program.

In most C++ programs, as we'll see later, `main()` calls member functions in various objects to carry out the program's real work. The `main()` function may also contain calls to other standalone functions. This is shown in Figure 2.1.

**FIGURE 2.1**

*Objects, functions, and `main()`.*

## Program Statements

The program *statement* is the fundamental unit of C++ programming. There are two statements in the FIRST program: the line

```
cout << "Every age has a language of its own\n";
```

and the return statement

```
return 0;
```

The first statement tells the computer to display the quoted phrase. Most statements tell the computer to do something. In this respect, statements in C++ are similar to statements in other languages. In fact, as we've noted, the majority of statements in C++ are identical to statements in C.

A semicolon signals the end of the statement. This is a crucial part of the syntax but easy to forget. In some languages (like BASIC), the end of a statement is signaled by the end of the line, but that's not true in C++. If you leave out the semicolon, the compiler will often (although not always) signal an error.

The last statement in the function body is `return 0;`. This tells `main()` to return the value 0 to whoever called it, in this case the operating system or compiler. In older versions of C++ you could give `main()` the return type of `void` and dispense with the `return` statement, but this is not considered correct in Standard C++. We'll learn more about `return` in Chapter 5.

## Whitespace

We mentioned that the end of a line isn't important to a C++ compiler. Actually, the compiler ignores whitespace almost completely. *Whitespace* is defined as spaces, carriage returns, line-feeds, tabs, vertical tabs, and formfeeds. These characters are invisible to the compiler. You can put several statements on one line, separated by any number of spaces or tabs, or you can run a statement over two or more lines. It's all the same to the compiler. Thus the FIRST program could be written this way:

```
#include <iostream>
using
namespace std;

int main () { cout
<<
"Every age has a language of its own\n"
; return
0;}
```

We don't recommend this syntax—it's nonstandard and hard to read—but it does compile correctly.

There are several exceptions to the rule that whitespace is invisible to the compiler. The first line of the program, starting with `#include`, is a preprocessor directive, which must be written on one line. Also, string constants, such as `"Every age has a language of its own"`, cannot be broken into separate lines. (If you need a long string constant, you can insert a backslash (`\`) at the line break or divide the string into two separate strings, each surrounded by quotes.)

## Output Using `cout`

As you have seen, the statement

```
cout << "Every age has a language of its own\n";
```

causes the phrase in quotation marks to be displayed on the screen. How does this work? A complete description of this statement requires an understanding of objects, operator overloading, and other topics we won't discuss until later in the book, but here's a brief preview.

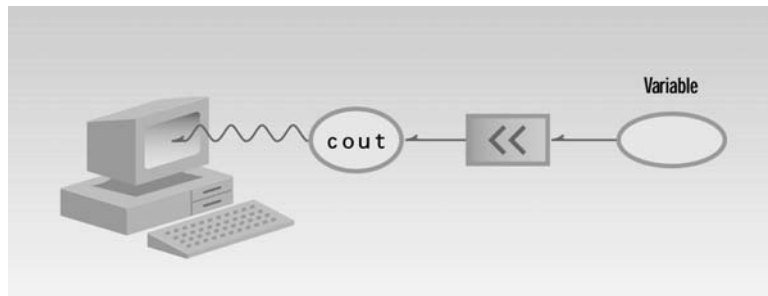


The identifier `cout` (pronounced “C out”) is actually an *object*. It is predefined in C++ to correspond to the *standard output stream*. A *stream* is an abstraction that refers to a flow of data. The standard output stream normally flows to the screen display—although it can be redirected to other output devices. We’ll discuss streams (and redirection) in Chapter 12, “Streams and Files.”

The operator `<<` is called the *insertion or put to operator*. It directs the contents of the variable on its right to the object on its left. In FIRST it directs the string constant “Every age has a language of its own\n” to `cout`, which sends it to the display.

(If you know C, you’ll recognize `<<` as the *left-shift* bit-wise operator and wonder how it can also be used to direct output. In C++, operators can be overloaded. That is, they can perform different activities, depending on the context. We’ll learn about overloading in Chapter 8, “Operator Overloading.”)

Although the concepts behind the use of `cout` and `<<` may be obscure at this point, using them is easy. They’ll appear in almost every example program. Figure 2.2 shows the result of using `cout` and the insertion operator `<<`.



**FIGURE 2.2**

*Output with cout.*

## String Constants

The phrase in quotation marks, “Every age has a language of its own\n”, is an example of a *string constant*. As you probably know, a constant, unlike a variable, cannot be given a new value as the program runs. Its value is set when the program is written, and it retains this value throughout the program’s existence.

As we’ll see later, the situation regarding strings is rather complicated in C++. Two ways of handling strings are commonly used. A string can be represented by an array of characters, or it can be represented as an object of a class. We’ll learn more about both kinds of strings in Chapter 7, “Arrays and Strings.”

The `'\n'` character at the end of the string constant is an example of an *escape sequence*. It causes the next text output to be displayed on a new line. We use it here so that the phrases such as “Press any key to continue,” inserted by some compilers for display after the program terminates, will appear on a new line. We’ll discuss escape sequences later in this chapter.

## Directives

The two lines that begin the `FIRST` program are *directives*. The first is a *preprocessor directive*, and the second is a *using directive*. They occupy a sort of gray area: They’re not part of the basic C++ language, but they’re necessary anyway

## Preprocessor Directives

The first line of the `FIRST` program

```
#include <iostream>
```

might look like a program statement, but it’s not. It isn’t part of a function body and doesn’t end with a semicolon, as program statements must. Instead, it starts with a number sign (`#`). It’s called a *preprocessor directive*. Recall that program statements are instructions to the *computer* to do something, such as adding two numbers or printing a sentence. A preprocessor directive, on the other hand, is an instruction to the *compiler*. A part of the compiler called the *preprocessor* deals with these directives before it begins the real compilation process.

The preprocessor directive `#include` tells the compiler to insert another file into your source file. In effect, the `#include` directive is replaced by the contents of the file indicated. Using an `#include` directive to insert another file into your source file is similar to pasting a block of text into a document with your word processor.

`#include` is only one of many preprocessor directives, all of which can be identified by the initial `#` sign. The use of preprocessor directives is not as common in C++ as it is in C, but we’ll look at a few additional examples as we go along. The type file usually included by `#include` is called a *header file*.

## Header Files

In the `FIRST` example, the preprocessor directive `#include` tells the compiler to add the source file `IOSTREAM` to the `FIRST.CPP` source file before compiling. Why do this? `IOSTREAM` is an example of a *header file* (sometimes called an *include file*). It’s concerned with basic input/output operations, and contains declarations that are needed by the `cout` identifier and the `<<` operator. Without these declarations, the compiler won’t recognize `cout` and will think `<<` is being used incorrectly. There are many such include files. The newer Standard C++ header files don’t have a file extension, but some older header files, left over from the days of the C language, have the extension `.H`.

If you want to see what's in `iostream`, you can find the `include` directory for your compiler and display it as a source file in the Edit window. (See the appropriate appendix for hints on how to do this.) Or you can look at it with the WordPad or Notepad utilities. The contents won't make much sense at this point, but you will at least prove to yourself that `iostream` is a source file, written in normal ASCII characters.

We'll return to the topic of header files at the end of this chapter, when we introduce library functions.

## The `using` Directive

A C++ program can be divided into different *namespaces*. A namespace is a part of the program in which certain names are recognized; outside of the namespace they're unknown. The directive

```
using namespace std;
```

says that all the program statements that follow are within the `std` namespace. Various program components such as `cout` are declared within this namespace. If we didn't use the `using` directive, we would need to add the `std` name to many program elements. For example, in the `FIRST` program we'd need to say

```
std::cout << "Every age has a language of its own.";
```

To avoid adding `std::` dozens of times in programs we use the `using` directive instead. We'll discuss namespaces further in Chapter 13, "Multifile Programs."

## Comments

Comments are an important part of any program. They help the person writing a program, and anyone else who must read the source file, understand what's going on. The compiler ignores comments, so they do not add to the file size or execution time of the executable program.

## Comment Syntax

Let's rewrite our `FIRST` program, incorporating comments into our source file. We'll call the new program `COMMENTS`:

```
// comments.cpp
// demonstrates comments
#include <iostream>           //preprocessor directive
using namespace std;        //"using" directive
```

```
int main()                //function name "main"
{                          //start function body
    cout << "Every age has a language of its own\n"; //statement
    return 0;              //statement
}                          //end function body
```

Comments start with a double slash symbol (//) and terminate at the end of the line. (This is one of the exceptions to the rule that the compiler ignores whitespace.) A comment can start at the beginning of the line or on the same line following a program statement. Both possibilities are shown in the COMMENTS example.

## When to Use Comments

Comments are almost always a good thing. Most programmers don't use enough of them. If you're tempted to leave out comments, remember that not everyone is as smart as you; they may need more explanation than you do about what your program is doing. Also, you may not be as smart next month, when you've forgotten key details of your program's operation, as you are today.

Use comments to explain to the person looking at the listing what you're trying to do. The details are in the program statements themselves, so the comments should concentrate on the big picture, clarifying your reasons for using a certain statement or group of statements.

## Alternative Comment Syntax

There's a second comment style available in C++:

```
/* this is an old-style comment */
```

This type of comment (the only comment originally available in C) begins with the /\* character pair and ends with \*/ (not with the end of the line). These symbols are harder to type (since / is lowercase while \* is uppercase) and take up more space on the line, so this style is not generally used in C++. However, it has advantages in special situations. You can write a multiline comment with only two comment symbols:

```
/* this
is a
potentially
very long
multiline
comment
*/
```

This is a good approach to making a comment out of a large text passage, since it saves inserting the // symbol on every line.

You can also insert a `/* */` comment anywhere within the text of a program line:

```
func1()  
{ /* empty function body */ }
```

If you attempt to use the `//` style comment in this case, the closing brace won't be visible to the compiler—since a `//` style comment runs to the end of the line—and the code won't compile correctly.

## Integer Variables

Variables are the most fundamental part of any language. A variable has a symbolic name and can be given a variety of values. Variables are located in particular places in the computer's memory. When a variable is given a value, that value is actually placed in the memory space assigned to the variable. Most popular languages use the same general variable types, such as integers, floating-point numbers, and characters, so you are probably already familiar with the ideas behind them.

Integer variables represent integer numbers like 1, 30,000, and -27. Such numbers are used for counting discrete numbers of objects, like 11 pencils or 99 bottles of beer. Unlike floating-point numbers, integers have no fractional part; you can express the idea of *four* using integers, but not *four and one-half*.

## Defining Integer Variables

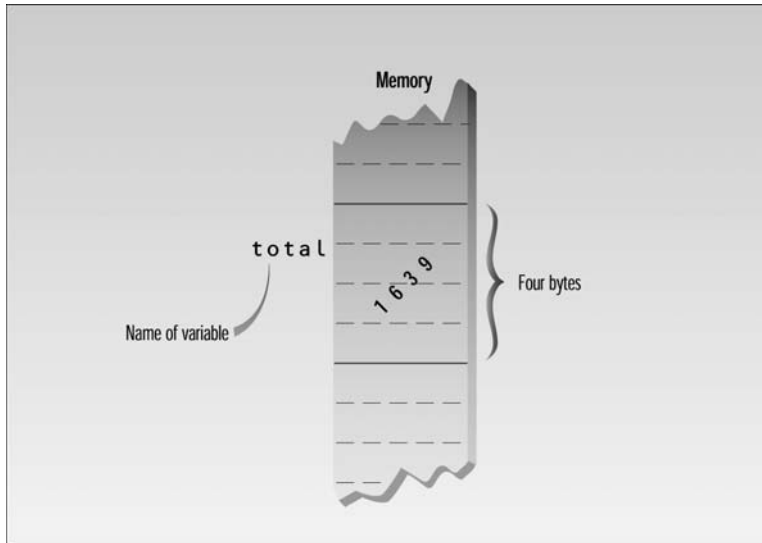
Integer variables exist in several sizes, but the most commonly used is type `int`. The amount of memory occupied by the integer types is system dependent. On a 32-bit system such as Windows, an `int` occupies 4 bytes (which is 32 bits) of memory. This allows an `int` to hold numbers in the range from -2,147,483,648 to 2,147,483,647. Figure 2.3 shows an integer variable in memory.

While type `int` occupies 4 bytes on current Windows computers, it occupied only 2 bytes in MS-DOS and earlier versions of Windows. The ranges occupied by the various types are listed in the header file `LIMITS`; you can also look them up using your compiler's help system.

Here's a program that defines and uses several variables of type `int`:

```
// intvars.cpp  
// demonstrates integer variables  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int var1;           //define var1  
    int var2;           //define var2
```

```
var1 = 20;           //assign value to var1
var2 = var1 + 10;    //assign value to var2
cout << "var1+10 is "; //output text
cout << var2 << endl; //output value of var2
return 0;
}
```

**FIGURE 2.3**

*Variable of type int in memory.*

Type this program into your compiler's edit screen (or load it from the Web site), compile and link it, and then run it. Examine the output window. The statements

```
int var1;
int var2;
```

define two integer variables, var1 and var2. The keyword `int` signals the type of variable. These statements, which are called *declarations*, must terminate with a semicolon, like other program statements.

You must declare a variable before using it. However, you can place variable declarations anywhere in a program. It's not necessary to declare variables before the first executable statement (as was necessary in C). However, it's probably more readable if commonly-used variables are located at the beginning of the program.

## Declarations and Definitions

Let's digress for a moment to note a subtle distinction between the terms *definition* and *declaration* as applied to variables.

A *declaration* introduces a variable's name (such as `var1`) into a program and specifies its type (such as `int`). However, if a declaration also sets aside memory for the variable, it is also called a *definition*. The statements

```
int var1;  
int var2;
```

in the `INTVARS` program are definitions, as well as declarations, because they set aside memory for `var1` and `var2`. We'll be concerned mostly with declarations that are also definitions, but later on we'll see various kinds of declarations that are not definitions.

## Variable Names

The program `INTVARS` uses variables named `var1` and `var2`. The names given to variables (and other program features) are called *identifiers*. What are the rules for writing identifiers? You can use upper- and lowercase letters, and the digits from 1 to 9. You can also use the underscore (`_`). The first character must be a letter or underscore. Identifiers can be as long as you like, but most compilers will only recognize the first few hundred characters. The compiler distinguishes between upper- and lowercase letters, so `Var` is not the same as `var` or `VAR`.

You can't use a C++ keyword as a variable name. A *keyword* is a predefined word with a special meaning. `int`, `class`, `if`, and `while` are examples of keywords. A complete list of keywords can be found in Appendix B, "C++ Precedence Table and Keywords," and in your compiler's documentation.

Many C++ programmers follow the convention of using all lowercase letters for variable names. Other programmers use a mixture of upper- and lowercase, as in `IntVar` or `dataCount`. Still others make liberal use of underscores. Whichever approach you use, it's good to be consistent throughout a program. Names in all uppercase are sometimes reserved for constants (see the discussion of `const` that follows). These same conventions apply to naming other program elements such as classes and functions.

A variable's name should make clear to anyone reading the listing the variable's purpose and how it is used. Thus `boilerTemperature` is better than something cryptic like `bT` or `t`.

## Assignment Statements

The statements

```
var1 = 20;  
var2 = var1 + 10;
```

assign values to the two variables. The equal sign (=), as you might guess, causes the value on the right to be assigned to the variable on the left. The = in C++ is equivalent to the := in Pascal or the = in BASIC. In the first line shown here, var1, which previously had no value, is given the value 20.

## Integer Constants

The number 20 is an *integer constant*. Constants don't change during the course of the program. An integer constant consists of numerical digits. There must be no decimal point in an integer constant, and it must lie within the range of integers.

In the second program line shown here, the plus sign (+) adds the value of var1 and 10, in which 10 is another constant. The result of this addition is then assigned to var2.

## Output Variations

The statement

```
cout << "var1+10 is ";
```

displays a string constant, as we've seen before. The next statement

```
cout << var2 << endl;
```

displays the value of the variable var2. As you can see in your console output window, the output of the program is

```
var1+10 is 30
```

Note that cout and the << operator know how to treat an integer and a string differently. If we send them a string, they print it as text. If we send them an integer, they print it as a number. This may seem obvious, but it is another example of operator overloading, a key feature of C++. (C programmers will remember that such functions as printf() need to be told not only the variable to be displayed, but the type of the variable as well, which makes the syntax far less intuitive.)

As you can see, the output of the two cout statements appears on the same line on the output screen. No linefeed is inserted automatically. If you want to start on a new line, you must insert a linefeed yourself. We've seen how to do this with the '\n' escape sequence. Now we'll see another way: using something called a *manipulator*.

## The endl Manipulator

The last cout statement in the INTVARS program ends with an unfamiliar word: endl. This causes a linefeed to be inserted into the stream, so that subsequent text is displayed on the next line. It has the same effect as sending the '\n' character, but is somewhat clearer. It's an



example of a *manipulator*. Manipulators are instructions to the output stream that modify the output in various ways; we'll see more of them as we go along. Strictly speaking, `endl` (unlike `'\n'`) also causes the output buffer to be flushed, but this happens invisibly so for most purposes the two are equivalent.

## Other Integer Types

There are several numerical integer types besides type `int`. The two most common types are `long` and `short`. (Strictly speaking type `char` is an integer type as well, but we'll cover it separately.) We noted that the size of type `int` is system dependent. In contrast, types `long` and `short` have fixed sizes no matter what system is used.

Type `long` always occupies four bytes, which is the same as type `int` on 32-bit Windows systems. Thus it has the same range, from  $-2,147,483,648$  to  $2,147,483,647$ . It can also be written as `long int`; this means the same as `long`. There's little point in using type `long` on 32-bit systems, since it's the same as `int`. However, if your program may need to run on a 16-bit system such as MS-DOS, or on older versions of Windows, specifying type `long` will guarantee a four-bit integer type. In 16-bit systems, type `int` has the same range as type `short`.

On all systems type `short` occupies two bytes, giving it a range of  $-32,768$  to  $32,767$ . There's probably not much point using type `short` on modern Windows systems unless it's important to save memory. Type `int`, although twice as large, is accessed faster than type `short`.

If you want to create a constant of type `long`, use the letter `L` following the numerical value, as in

```
longvar = 7678L; // assigns long constant 7678 to longvar
```

Many compilers offer integer types that explicitly specify the number of bits used. (Remember there are 8 bits to a byte.) These type names are preceded by two underscores. They are `__int8`, `__int16`, `__int32`, and `__int64`. The `__int8` type corresponds to `char`, and (at least in 32-bit systems) the type name `__int16` corresponds to `short` and `__int32` corresponds to both `int` and `long`. The `__int64` type holds huge integers with up to 19 decimal digits. Using these type names has the advantage that the number of bytes used for a variable is not implementation dependent. However, this is not usually an issue, and these types are seldom used.

## Character Variables

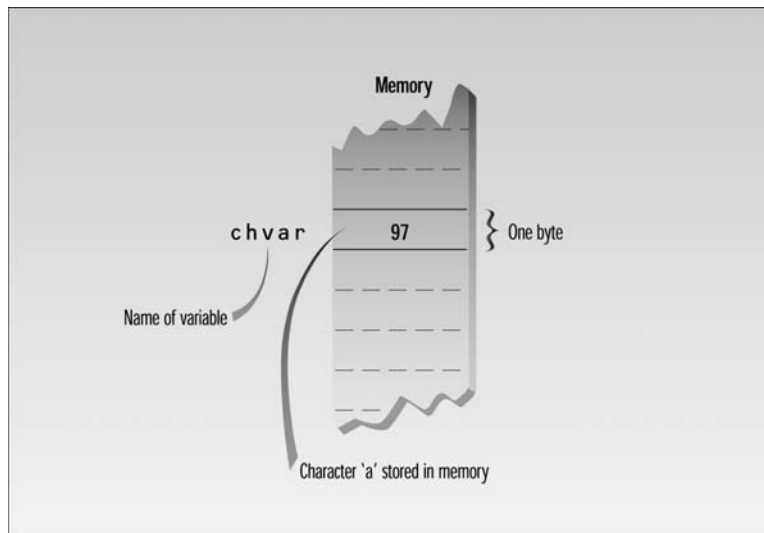
Type `char` stores integers that range in value from  $-128$  to  $127$ . Variables of this type occupy only 1 byte (eight bits) of memory. Character variables are sometimes used to store numbers that confine themselves to this limited range, but they are much more commonly used to store ASCII characters.

As you may already know, the ASCII character set is a way of representing characters such as 'a', 'B', '\$', '3', and so on, as numbers. These numbers range from 0 to 127. Most Windows systems extend this range to 255 to accommodate various foreign-language and graphics characters. Appendix A, “ASCII Table,” shows the ASCII character set.

Complexities arise when foreign languages are used, and even when programs are transferred between computer systems in the same language. This is because the characters in the range 128 to 255 aren't standardized and because the one-byte size of type `char` is too small to accommodate the number of characters in many languages, such as Japanese. **Standard C++ provides a larger character type called `wchar_t` to handle foreign languages.** This is important if you're writing programs for international distribution. However, in this book we'll ignore type `wchar_t` and assume that we're dealing with the ASCII character set found in current versions of Windows.

## Character Constants

Character constants use single quotation marks around a character, like 'a' and 'b'. (Note that this differs from *string* constants, which use double quotation marks.) When the C++ compiler encounters such a character constant, it translates it into the corresponding ASCII code. The constant **'a' appearing in a program, for example, will be translated into 97,** as shown in Figure 2.4.



**FIGURE 2.4**

*Variable of type `char` in memory.*

Character variables can be assigned character constants as values. The following program shows some examples of character constants and variables.

```
// charvars.cpp
// demonstrates character variables
#include <iostream>          //for cout, etc.
using namespace std;

int main()
{
    char charvar1 = 'A';    //define char variable as character
    char charvar2 = '\t';  //define char variable as tab

    cout << charvar1;      //display character
    cout << charvar2;      //display character
    charvar1 = 'B';        //set char variable to char constant
    cout << charvar1;      //display character
    cout << '\n';          //display newline character
    return 0;
}
```

## Initialization

Variables can be initialized at the same time they are defined. In this program two variables of type `char`—`charvar1` and `charvar2`—are initialized to the character constants `'A'` and `'\t'`.

## Escape Sequences

This second character constant, `'\t'`, is an odd one. Like `'\n'`, which we encountered earlier, it's an example of an *escape sequence*. The name reflects the fact that the backslash causes an “escape” from the normal way characters are interpreted. In this case the `t` is interpreted not as the character `'t'` but as the tab character. A tab causes printing to continue at the next tab stop. In console-mode programs, tab stops are positioned every eight spaces. Another character constant, `'\n'`, is sent directly to `cout` in the last line of the program.

Escape sequences can be used as separate characters or embedded in string constants. Table 2.1 shows a list of common escape sequences.

**TABLE 2.1** Common Escape Sequences

<i>Escape Sequence</i>	<i>Character</i>
<code>\ a</code>	Bell (beep)
<code>\ b</code>	Backspace
<code>\ f</code>	Formfeed

TABLE 2.1 Continued

<i>Escape Sequence</i>	<i>Character</i>
<code>\ n</code>	Newline
<code>\ r</code>	Return
<code>\ t</code>	Tab
<code>\ \</code>	Backslash
<code>\ '</code>	Single quotation mark
<code>\ "</code>	Double quotation marks
<code>\ xdd</code>	Hexadecimal notation

Since the backslash, the single quotation marks, and the double quotation marks all have specialized meanings when used in constants, they must be represented by escape sequences when we want to display them as characters. Here’s an example of a quoted phrase in a string constant:

```
cout << "\"Run, Spot, run,\" she said.\"";
```

This translates to

```
"Run, Spot, run," she said.
```

Sometimes you need to represent a character constant that doesn’t appear on the keyboard, such as the graphics characters above ASCII code 127. To do this, you can use the ‘\xdd’ representation, where each d stands for a hexadecimal digit. If you want to print a solid rectangle, for example, you’ll find such a character listed as decimal number 178, which is hexadecimal number B2 in the ASCII table. This character would be represented by the character constant ‘\xB2’. We’ll see some examples of this later.

The CHARVARS program prints the value of charvar1 (‘A’) and the value of charvar2 (a tab). It then sets charvar1 to a new value (‘B’), prints that, and finally prints the newline. The output looks like this:

```
A      B
```

# Input with cin

Now that we’ve seen some variable types in use, let’s see how a program accomplishes input. The next example program asks the user for a temperature in degrees Fahrenheit, converts it to Celsius, and displays the result. It uses integer variables.

```
// fahrenheit.cpp
// demonstrates cin, newline
#include <iostream>
using namespace std;

int main()
{
    int ftemp; //for temperature in fahrenheit

    cout << "Enter temperature in fahrenheit: ";
    cin >> ftemp;
    int ctemp = (ftemp-32) * 5 / 9;
    cout << "Equivalent in Celsius is: " << ctemp << '\n';
    return 0;
}
```

The statement

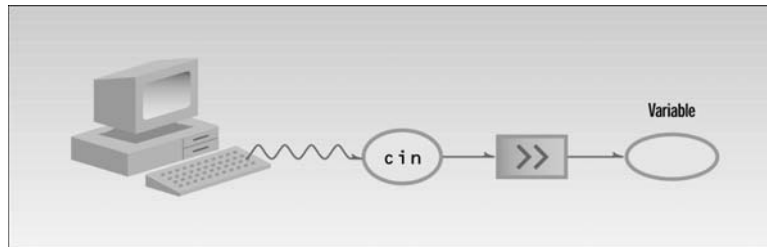
```
cin >> ftemp;
```

causes the program to wait for the user to type in a number. The resulting number is placed in the variable `ftemp`. The keyword `cin` (pronounced “C in”) is an object, predefined in C++ to correspond to the standard input stream. This stream represents data coming from the keyboard (unless it has been redirected). The `>>` is the *extraction or get from operator*. It takes the value from the stream object on its left and places it in the variable on its right.

Here’s some sample interaction with the program:

```
Enter temperature in fahrenheit: 212
Equivalent in Celsius is: 100
```

Figure 2.5 shows input using `cin` and the extraction operator `>>`.



**FIGURE 2.5**

*Input with cin.*

## Variables Defined at Point of Use

The FAHREN program has several new wrinkles besides its input capability. Look closely at the listing. Where is the variable `ctemp` defined? Not at the beginning of the program, but in the next-to-the-last line, where it's used to store the result of the arithmetic operation. As we noted earlier, you can define variables throughout a program, not just at the beginning. (Many languages, including C, require all variables to be defined before the first executable statement.)

Defining variables where they are used can make the listing easier to understand, since you don't need to refer repeatedly to the start of the listing to find the variable definitions. However, the practice should be used with discretion. Variables that are used in many places in a function are better defined at the start of the function.

## Cascading <<

The insertion operator `<<` is used repeatedly in the second `cout` statement in FAHREN. This is perfectly legal. The program first sends the phrase *Equivalent in Celsius is:* to `cout`, then it sends the value of `ctemp`, and finally the newline character `'\n'`.

The extraction operator `>>` can be cascaded with `cin` in the same way, allowing the user to enter a series of values. However, this capability is not used so often, since it eliminates the opportunity to prompt the user between inputs.

## Expressions

Any arrangement of variables, constants, and operators that specifies a computation is called an *expression*. Thus, `alpha+12` and `(alpha-37)*beta/2` are expressions. When the computations specified in the expression are performed, the result is usually a value. Thus if `alpha` is 7, the first expression shown has the value 19.

Parts of expressions may also be expressions. In the second example, `alpha-37` and `beta/2` are expressions. Even single variables and constants, like `alpha` and `37`, are considered to be expressions.

Note that expressions aren't the same as statements. Statements tell the compiler to do something and terminate with a semicolon, while expressions specify a computation. There can be several expressions in a statement.

## Precedence

Note the parentheses in the expression

```
(ftemp-32) * 5 / 9
```

Without the parentheses, the multiplication would be carried out first, since `*` has higher priority than `-`. With the parentheses, the subtraction is done first, then the multiplication, since all operations inside parentheses are carried out first. What about the precedence of the `*` and `/` signs? When two arithmetic operators have the same precedence, the one on the left is executed first, so in this case the multiplication will be carried out next, then the division.

Precedence and parentheses are normally applied this same way in algebra and in other computer languages, so their use probably seems quite natural. However, precedence is an important topic in C++. We'll return to it later when we introduce different kinds of operators.

## Floating Point Types

We've talked about type `int` and type `char`, both of which represent numbers as integers—that is, numbers without a fractional part. Now let's examine a different way of storing numbers—as floating-point variables.

Floating-point variables represent numbers with a decimal place—like 3.1415927, 0.0000625, and  $-10.2$ . They have both an integer part, to the left of the decimal point, and a fractional part, to the right. Floating-point variables represent what mathematicians call *real numbers*, which are used for measurable quantities such as distance, area, and temperature. They typically have a fractional part.

There are three kinds of floating-point variables in C++: type `float`, type `double`, and type `long double`. Let's start with the smallest of these, type `float`.

### Type `float`

Type `float` stores numbers in the range of about  $3.4 \times 10^{-38}$  to  $3.4 \times 10^{38}$ , with a precision of seven digits. It occupies 4 bytes (32 bits) in memory, as shown in Figure 2.6.

The following example program prompts the user to type in a floating-point number representing the radius of a circle. It then calculates and displays the circle's area.

```
// circarea.cpp
// demonstrates floating point variables
#include <iostream> //for cout, etc.
using namespace std;

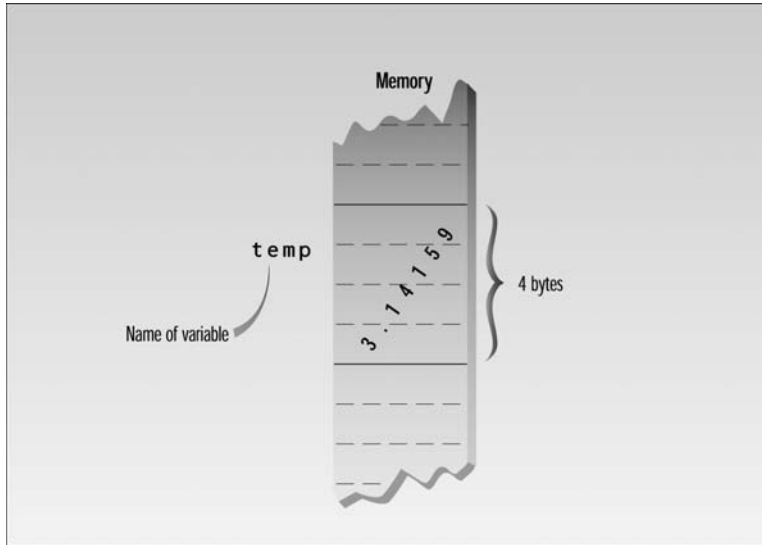
int main()
{
    float rad; //variable of type float
    const float PI = 3.14159F; //type const float

    cout << "Enter radius of circle: "; //prompt
    cin >> rad; //get radius
```

```

float area = PI * rad * rad;           //find area
cout << "Area is " << area << endl; //display answer
return 0;
}

```

**FIGURE 2.6**

*Variable of type float in memory.*

Here's a sample interaction with the program:

```

Enter radius of circle: 0.5
Area is 0.785398

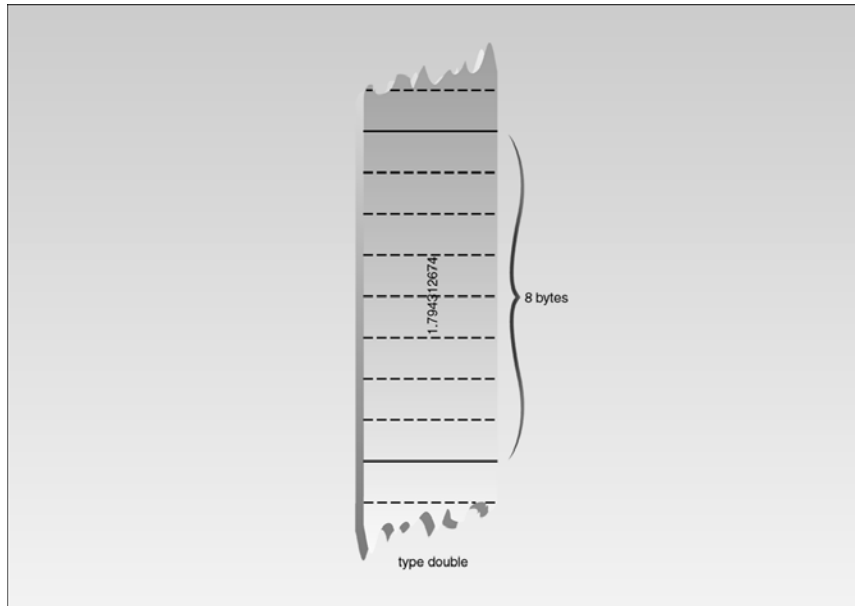
```

This is the area in square feet of a 12-inch LP record (which has a radius of 0.5 feet). At one time this was an important quantity for manufacturers of vinyl.

## Type double and long double

The larger floating point types, `double` and `long double`, are similar to `float` except that they require more memory space and provide a wider range of values and more precision. Type `double` requires 8 bytes of storage and handles numbers in the range from  $1.7 \times 10^{-308}$  to  $1.7 \times 10^{308}$  with a precision of 15 digits. Type `long double` is compiler-dependent but is often the same as `double`. Type `double` is shown in Figure 2.7.



**FIGURE 2.7**

Variable of type `double`.

## Floating-Point Constants

The number `3.14159F` in `CIRCAREA` is an example of a *floating-point constant*. The decimal point signals that it is a floating-point constant, and not an integer, and the `F` specifies that it's type `float`, rather than `double` or `long double`. The number is written in normal decimal notation. You don't need a suffix letter with constants of type `double`; it's the default. With type `long double`, use the letter `L`.

You can also write floating-point constants using *exponential notation*. Exponential notation is a way of writing large numbers without having to write out a lot of zeros. For example, 1,000,000,000 can be written as `1.0E9` in exponential notation. Similarly, `1234.56` would be written `1.23456E3`. (This is the same as 1.23456 times  $10^3$ .) The number following the `E` is called the *exponent*. It indicates how many places the decimal point must be moved to change the number to ordinary decimal notation.

The exponent can be positive or negative. The exponential number `6.35239E-5` is equivalent to 0.0000635239 in decimal notation. This is the same as 6.35239 times  $10^{-5}$ .

## The const Qualifier

Besides demonstrating variables of type `float`, the `CIRCAREA` example also introduces the qualifier `const`. It's used in the statement

```
const float PI = 3.14159F; //type const float
```

The keyword `const` (for constant) precedes the data type of a variable. It specifies that the value of a variable will not change throughout the program. Any attempt to alter the value of a variable defined with this qualifier will elicit an error message from the compiler.

The qualifier `const` ensures that your program does not inadvertently alter a variable that you intended to be a constant, such as the value of `PI` in `CIRCAREA`. It also reminds anyone reading the listing that the variable is not intended to change. The `const` modifier can apply to other entities besides simple variables. We'll learn more about this as we go along.

## The #define Directive

Although the construction is not recommended in C++, constants can also be specified using the preprocessor directive `#define`. This directive sets up an equivalence between an identifier and a text phrase. For example, the line

```
#define PI 3.14159
```

appearing at the beginning of your program specifies that the identifier `PI` will be replaced by the text `3.14159` throughout the program. This construction has long been popular in C.

However, you can't specify the data type of the constant using `#define`, which can lead to program bugs; so even in C `#define` has been superseded by `const` used with normal variables. However, you may encounter this construction in older programs.

## Type bool

For completeness we should mention type `bool` here, although it won't be important until we discuss relational operators in the next chapter.

We've seen that variables of type `int` can have billions of possible values, and those of type `char` can have 256. Variables of type `bool` can have only two possible values: `true` and `false`. In theory a `bool` type requires only one bit (not byte) of storage, but in practice compilers often store them as bytes because a byte can be quickly accessed, while an individual bit must be extracted from a byte, which requires additional time.

As we'll see, type `bool` is most commonly used to hold the results of comparisons. Is alpha less than beta? If so, a `bool` value is given the value `true`; if not, it's given the value `false`.

Type `bool` gets its name from George Boole, a 19th century English mathematician who invented the concept of using logical operators with true-or-false values. Thus such true/false values are often called *Boolean* values.

## The `setw` Manipulator

We've mentioned that manipulators are operators used with the insertion operator (`<<`) to modify—or manipulate—the way data is displayed. We've already seen the `endl` manipulator; now we'll look at another one: `setw`, which changes the field width of output.

You can think of each value displayed by `cout` as occupying a field: an imaginary box with a certain width. The default field is just wide enough to hold the value. That is, the integer 567 will occupy a field three characters wide, and the string "pajamas" will occupy a field seven characters wide. However, in certain situations this may not lead to optimal results. Here's an example. The `WIDTH1` program prints the names of three cities in one column, and their populations in another.

```
// width1.cpp
// demonstrates need for setw manipulator
#include <iostream>
using namespace std;

int main()
{
    long pop1=2425785, pop2=47, pop3=9761;

    cout << "LOCATION " << "POP." << endl
         << "Portcity " << pop1 << endl
         << "Hightown " << pop2 << endl
         << "Lowville " << pop3 << endl;
    return 0;
}
```

Here's the output from this program:

```
LOCATION POP.
Portcity 2425785
Hightown 47
Lowville 9761
```

Unfortunately, this format makes it hard to compare the numbers; it would be better if they lined up to the right. Also, we had to insert spaces into the names of the cities to separate them from the numbers. This is an inconvenience.

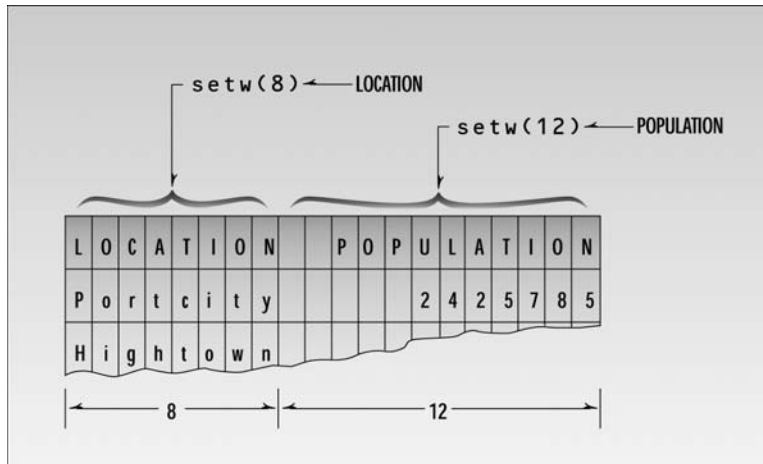
Here's a variation of this program, `WIDTH2`, that uses the `setw` manipulator to eliminate these problems by specifying field widths for the names and the numbers:

```
// width2.cpp
// demonstrates setw manipulator
#include <iostream>
#include <iomanip>    // for setw
using namespace std;

int main()
{
    long pop1=2425785, pop2=47, pop3=9761;

    cout << setw(8) << "LOCATION" << setw(12)
         << "POPULATION" << endl
         << setw(8) << "Portcity" << setw(12) << pop1 << endl
         << setw(8) << "Hightown" << setw(12) << pop2 << endl
         << setw(8) << "Lowville" << setw(12) << pop3 << endl;
    return 0;
}
```

The `setw` manipulator causes the number (or string) that follows it in the stream to be printed within a field *n* characters wide, where *n* is the argument to `setw(n)`. The value is right-justified within the field. Figure 2.8 shows how this looks. Type `long` is used for the population figures, which prevents a potential overflow problem on systems that use 2-byte integer types, in which the largest integer value is 32,767.



**FIGURE 2.8**

*Field widths and setw.*

Here's the output of WIDTH2:

```
LOCATION  POPULATION
Portcity    2425785
Hightown    47
Lowville    9761
```

## Cascading the Insertion Operator

Note that there's only one `cout` statement in WIDTH1 and WIDTH2, although it's written on multiple lines. In doing this, we take advantage of the fact that the compiler ignores whitespace, and that the insertion operator can be cascaded. The effect is the same as using four separate statements, each beginning with `cout`.

## Multiple Definitions

We initialized the variables `pop1`, `pop2`, and `pop3` to specific values at the same time we defined them. This is similar to the way we initialized `char` variables in the CHARVARS example. Here, however, we've defined and initialized all three variables on one line, using the same long keyword and separating the variable names with commas. This saves space where a number of variables are all the same type.

## The IOMANIP Header File

The declarations for the manipulators (except `endl`) are not in the usual `iostream` header file, but in a separate header file called `IOMANIP`. When you use these manipulators you must `#include` this header file in your program, as we do in the WIDTH2 example.

## Variable Type Summary

Our program examples so far have used four data types—`int`, `char`, `float`, and `long`. In addition we've mentioned types `bool`, `short`, `double`, and `long double`. Let's pause now to summarize these data types. Table 2.2 shows the keyword used to define the type, the numerical range the type can accommodate, the digits of precision (in the case of floating-point numbers), and the bytes of memory occupied in a 32-bit environment.

**TABLE 2.2** Basic C++ Variable Types

Keyword	Numerical Range		Digits of Precision	Bytes of Memory
	Low	High		
<code>bool</code>	false	true	n/a	1
<code>char</code>	−128	127	n/a	1
<code>short</code>	−32,768	32,767	n/a	2

TABLE 2.2 Continued

Keyword	Numerical Range		Digits of Precision	Bytes of Memory
	Low	High		
int	−2,147,483,648	2,147,483,647	n/a	4
long	−2,147,483,648	2,147,483,647	n/a	4
float	3.4 x 10 <sup>−38</sup>	3.4 x 10 <sup>38</sup>	7	4
double	1.7 x 10 <sup>−308</sup>	1.7 x 10 <sup>308</sup>	15	8

unsigned Data Types

By eliminating the sign of the character and integer types, you can change their range to start at 0 and include only positive numbers. This allows them to represent numbers twice as big as the signed type. Table 2.3 shows the unsigned versions.

TABLE 2.3 Unsigned Integer Types

Keyword	Numerical Range		Bytes of Memory
	Low	High	
unsigned char	0	255	1
unsigned short	0	65,535	2
unsigned int	0	4,294,967,295	4
unsigned long	0	4,294,967,295	4

The unsigned types are used when the quantities represented are always positive—such as when representing a count of something—or when the positive range of the signed types is not quite large enough.

To change an integer type to an unsigned type, precede the data type keyword with the keyword `unsigned`. For example, an unsigned variable of type `char` would be defined as

```
unsigned char ucharvar;
```

Exceeding the range of signed types can lead to obscure program bugs. In certain (probably rare) situations such bugs can be eliminated by using unsigned types. For example, the following program stores the constant 1,500,000,000 (1.5 billion) both as an `int` in `signedVar` and as an unsigned `int` in `unsignVar`.

```
// signtest.cpp
// tests signed and unsigned integers
#include <iostream>
```

```
using namespace std;

int main()
{
    int signedVar = 1500000000;           //signed
    unsigned int unsignVar = 1500000000;  //unsigned

    signedVar = (signedVar * 2) / 3;  //calculation exceeds range
    unsignVar = (unsignVar * 2) / 3;  //calculation within range

    cout << "signedVar = " << signedVar << endl;  //wrong
    cout << "unsignVar = " << unsignVar << endl;  //OK
    return 0;
}
```

The program multiplies both variables by 2, then divides them by 3. Although the result is smaller than the original number, the intermediate calculation is larger than the original number. This is a common situation, but it can lead to trouble. In `SIGNTEST` we expect that two-thirds the original value, or 1,000,000,000, will be restored to both variables. Unfortunately, in `signedVar` the multiplication created a result—3,000,000,000—that exceeded the range of the `int` variable (−2,147,483,648 to 2,147,483,647). Here’s the output:

```
signedVar = -431,655,765
unsignVar = 1,000,000,000
```

The signed variable now displays an incorrect answer, while the unsigned variable, which is large enough to hold the intermediate result of the multiplication, records the result correctly. The moral is this: Be careful that all values generated in your program are within the range of the variables that hold them. (The results will be different on 16-bit or 64-bit computers, which use different numbers of bytes for type `int`.)

## Type Conversion

C++, like C, is more forgiving than some languages in the way it treats expressions involving several different data types. As an example, consider the `MIXED` program:

```
// mixed.cpp
// shows mixed expressions
#include <iostream>
using namespace std;

int main()
{
    int count = 7;
    float avgWeight = 155.5F;
```

```
double totalWeight = count * avgWeight;
cout << "totalWeight=" << totalWeight << endl;
return 0;
}
```

Here a variable of type `int` is multiplied by a variable of type `float` to yield a result of type `double`. This program compiles without error; the compiler considers it normal that you want to multiply (or perform any other arithmetic operation on) numbers of different types.

Not all languages are this relaxed. Some don't permit mixed expressions, and would flag the line that performs the arithmetic in `MIXED` as an error. Such languages assume that when you mix types you're making a mistake, and they try to save you from yourself. C++ and C, however, assume that you must have a good reason for doing what you're doing, and they help carry out your intentions. This is one reason for the popularity of C++ and C. They give you more freedom. Of course, with more freedom, there's also more opportunity for you to make a mistake.

### Automatic Conversions

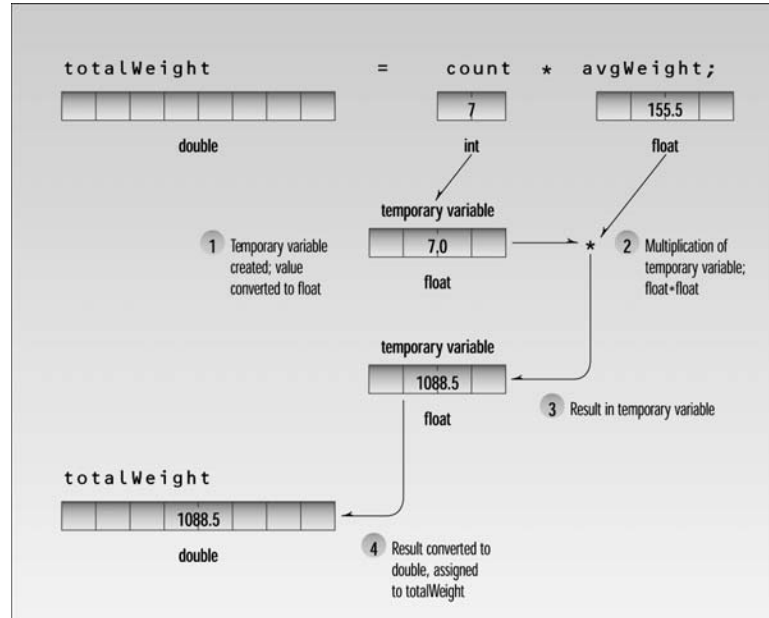
Let's consider what happens when the compiler confronts such mixed-type expressions as the one in `MIXED`. Types are considered "higher" or "lower," based roughly on the order shown in Table 2.4.

**TABLE 2.4** Order of Data Types

<i>Data Type</i>	<i>Order</i>
long double	Highest
double	
float	
long	
int	
short	
char	Lowest

The arithmetic operators such as `+` and `*` like to operate on two operands of the same type. When two operands of different types are encountered in the same expression, the lower-type variable is converted to the type of the higher-type variable. Thus in `MIXED`, the `int` value of `count` is converted to type `float` and stored in a temporary variable before being multiplied by the `float` variable `avgWeight`. The result (still of type `float`) is then converted to `double` so that it can be assigned to the `double` variable `totalWeight`. This process is shown in Figure 2.9.



**FIGURE 2.9***Data conversion.*

These conversions take place invisibly, and ordinarily you don't need to think too much about them; C++ automatically does what you want. However, sometimes the compiler isn't so happy about conversions, as we'll see in a moment. Also, when we start to use objects, we will in effect be defining our own data types. We may want to use these new data types in mixed expressions, just as we use normal variables in mixed expressions. When this is the case, we must be careful to create our own conversion routines to change objects of one type into objects of another. The compiler won't do it for us, as it does here with the built-in data types.

## Casts

*Casts* sounds like something to do with social classes in India, but in C++ the term applies to data conversions specified by the programmer, as opposed to the automatic data conversions we just described. Casts are also called *type casts*. What are casts for? Sometimes a programmer needs to convert a value from one type to another in a situation where the compiler will not do it automatically or without complaining.

There are several kinds of casts in Standard C++: static casts, dynamic casts, reinterpret casts, and `const` casts. Here we'll be concerned only with static casts; we'll learn about the others, which are used in more specialized situations, in later chapters.

C++ casts have a rather forbidding appearance. Here's a statement that uses a C++ cast to change a variable of type `int` into a variable of type `char`:

```
aCharVar = static_cast<char>(anIntVar);
```

Here the variable to be cast (`anIntVar`) is placed in parentheses and the type it's to be changed to (`char`) is placed in angle brackets. The result is that `anIntVar` is changed to type `char` before it's assigned to `aCharVar`. In this case the assignment statement would have carried out the cast itself, but there are situations where the cast is essential.

Recall that in the `SIGTEST` example an intermediate result exceeded the capacity of the variable type, resulting in an erroneous result. We fixed the problem by using `unsigned int` instead of `int`. This worked because the intermediate result—3,000,000,000—would fit in the range of the unsigned variable.

But suppose an intermediate result won't fit the unsigned type either. In such a case we might be able to solve the problem by using a cast. Here's an example:

```
// cast.cpp
// tests signed and unsigned integers
#include <iostream>
using namespace std;

int main()
{
    int intVar = 1500000000;           //1,500,000,000
    intVar = (intVar * 10) / 10;       //result too large
    cout << "intVar = " << intVar << endl; //wrong answer

    intVar = 1500000000;              //cast to double
    intVar = (static_cast<double>(intVar) * 10) / 10;
    cout << "intVar = " << intVar << endl; //right answer
    return 0;
}
```

When we multiply the variable `intVar` by 10, the result—15,000,000,000—is far too large to fit in a variable of type `int` or `unsigned int`. This leads to the wrong answer, as shown by the output of the first part of the program.

We could redefine the data type of the variables to be `double`; this provides plenty of room, since this type holds numbers with up to 15 digits. But suppose that for some reason, such as keeping the program small, we don't want to change the variables to type `double`. In this case there's another solution: We can cast `intVar` to type `double` before multiplying. This is sometimes called *coercion*; the data is coerced into becoming another type. The expression

```
static_cast<double>(intVar)
```

casts `intVar` to type `double`. It generates a temporary variable of type `double` with the same value as `intVar`. It is this temporary variable that is multiplied by 10. Since it is type `double`, the result fits. This result is then divided by 10 and assigned to the normal `int` variable `intVar`. Here's the program's output:

```
intVar = 211509811
intVar = 1500000000
```

The first answer, without the cast, is wrong; but in the second answer, the cast produces the correct result.

Before Standard C++, casts were handled using quite a different format. Instead of

```
aCharVar = static_cast<char>(anIntVar);
```

you could say

```
aCharVar = (char)anIntVar;
```

or alternatively

```
aCharVar = char(anIntVar);
```

One problem with these approaches is that they are hard to see; the syntax blends into the rest of the listing. They are also hard to search for using a Find operation with your source code editor. The new format solves this problem: `static_cast` is easy to see and easy to search for. These old casts still work, but their use is discouraged (or *deprecated*, to use the technical term).

Casts should be used only when absolutely necessary. They are a controlled way of evading *type safety* (which means making sure that variables don't change types by mistake) and can lead to trouble because they make it impossible for the compiler to spot potential problems. However, sometimes casts can't be avoided. We'll see some examples of situations where casts are necessary as we go along.

## Arithmetic Operators

As you have probably gathered by this time, C++ uses the four normal arithmetic operators `+`, `-`, `*`, and `/` for addition, subtraction, multiplication, and division. These operators work on all the data types, both integer and floating-point. They are used in much the same way that they are used in other languages, and are closely analogous to their use in algebra. However, there are some other arithmetic operators whose use is not so obvious.

## The Remainder Operator

There is a fifth arithmetic operator that works only with integer variables (types `char`, `short`, `int`, and `long`). It's called the *remainder operator*, and is represented by the percent symbol (%). This operator (also called the *modulus operator*) finds the remainder when one number is divided by another. The `REMAIND` program demonstrates the effect.

```
// remaind.cpp
// demonstrates remainder operator
#include <iostream>
using namespace std;

int main()
{
    cout << 6 % 8 << endl    // 6
         << 7 % 8 << endl    // 7
         << 8 % 8 << endl    // 0
         << 9 % 8 << endl    // 1
         << 10 % 8 << endl;   // 2
    return 0;
}
```

Here the numbers 6–10 are divided by 8, using the remainder operator. The answers are 6, 7, 0, 1, and 2—the remainders of these divisions. The remainder operator is used in a wide variety of situations. We'll show examples as we go along.

A note about precedence: In the expression

```
cout << 6 % 8
```

the remainder operator is evaluated first because it has higher precedence than the `<<` operator. If it did not, we would need to put parentheses around `6 % 8` to ensure it was evaluated before being acted on by `<<`.

## Arithmetic Assignment Operators

C++ offers several ways to shorten and clarify your code. One of these is the *arithmetic assignment operator*. This operator helps to give C++ listings their distinctive appearance.

The following kind of statement is common in most languages.

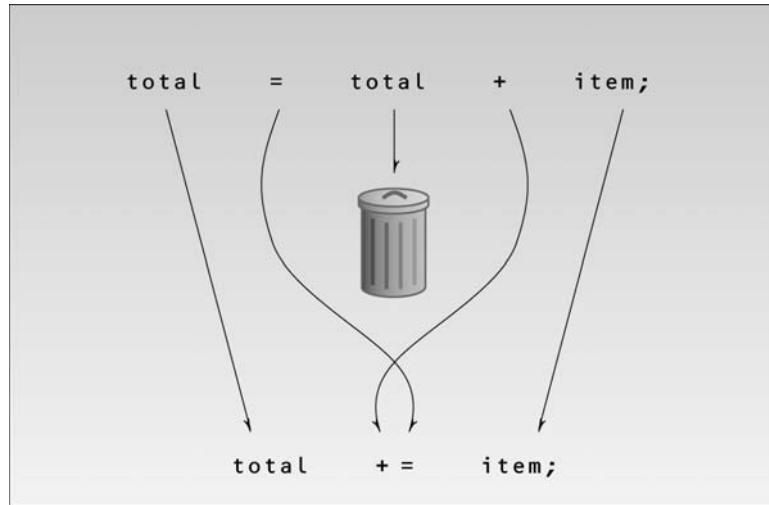
```
total = total + item; // adds "item" to "total"
```

In this situation you add something to an existing value (or you perform some other arithmetic operation on it). But the syntax of this statement offends those for whom brevity is important, because the name `total` appears twice. So C++ offers a condensed approach: the arithmetic assignment operator, which combines an arithmetic operator and an assignment operator and

eliminates the repeated operand. Here's a statement that has exactly the same effect as the preceding one.

```
total += item;    // adds "item" to "total"
```

Figure 2.10 emphasizes the equivalence of the two forms.



**FIGURE 2.10**

*Arithmetic assignment operator.*

There are arithmetic assignment operators corresponding to all the arithmetic operations: `+=`, `-=`, `*=`, `/=`, and `%=` (and some other operators as well). The following example shows the arithmetic assignment operators in use:

```
// assign.cpp
// demonstrates arithmetic assignment operators
#include <iostream>
using namespace std;

int main()
{
    int ans = 27;

    ans += 10;           //same as: ans = ans + 10;
    cout << ans << " ";
    ans -= 7;           //same as: ans = ans - 7;
    cout << ans << " ";
    ans *= 2;           //same as: ans = ans * 2;
    cout << ans << " ";
```

```
ans /= 3;           //same as: ans = ans / 3;
cout << ans << ", ";
ans %= 3;          //same as: ans = ans % 3;
cout << ans << endl;
return 0;
}
```

Here's the output from this program:

37, 30, 60, 20, 2

You don't need to use arithmetic assignment operators in your code, but they are a common feature of the language; they'll appear in numerous examples in this book.

## Increment Operators

Here's an even more specialized operator. You often need to add 1 to the value of an existing variable. You can do this the "normal" way:

```
count = count + 1;    // adds 1 to "count"
```

Or you can use an arithmetic assignment operator:

```
count += 1;    // adds 1 to "count"
```

But there's an even more condensed approach:

```
++count;    // adds 1 to "count"
```

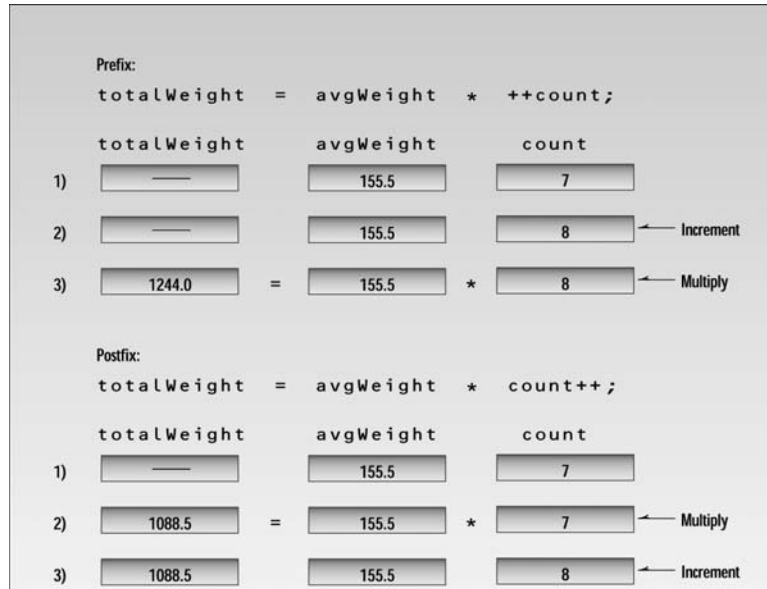
The ++ operator increments (adds 1 to) its argument.

### Prefix and Postfix

As if this weren't weird enough, the increment operator can be used in two ways: as a *prefix*, meaning that the operator precedes the variable; and as a *postfix*, meaning that the operator follows the variable. What's the difference? Often a variable is incremented within a statement that performs some other operation on it. For example

```
totalWeight = avgWeight * ++count;
```

The question here is this: Is the multiplication performed before or after count is incremented? In this case count is incremented first. How do we know that? Because prefix notation is used: ++count. If we had used postfix notation, count++, the multiplication would have been performed first, then count would have been incremented. This is shown in Figure 2.11.

**FIGURE 2.11***The increment operator.*

Here's an example that shows both the prefix and postfix versions of the increment operator:

```
// increm.cpp
// demonstrates the increment operator
#include <iostream>
using namespace std;

int main()
{
    int count = 10;

    cout << "count=" << count << endl;    //displays 10
    cout << "count=" << ++count << endl;    //displays 11 (prefix)
    cout << "count=" << count << endl;    //displays 11
    cout << "count=" << count++ << endl;    //displays 11 (postfix)
    cout << "count=" << count << endl;    //displays 12
    return 0;
}
```

Here's the program's output:

```
count=10
count=11
```

```
count=11
count=11
count=12
```

The first time `count` is incremented, the prefix `++` operator is used. This causes the increment to happen at the beginning of the statement evaluation, before the output operation has been carried out. When the value of the expression `++count` is displayed, it has already been incremented, and `<<` sees the value 11. The second time `count` is incremented, the postfix `++` operator is used. When the expression `count++` is displayed, it retains its unincremented value of 11. Following the completion of this statement, the increment takes effect, so that in the last statement of the program we see that `count` has acquired the value 12.

## The Decrement (--) Operator

The decrement operator, `--`, behaves very much like the increment operator, except that it subtracts 1 from its operand. It too can be used in both prefix and postfix forms.

## Library Functions

Many activities in C++ are carried out by *library functions*. These functions perform file access, mathematical computations, and data conversion, among other things. We don't want to dig too deeply into library functions before we explain how functions work (see Chapter 5), but you can use simple library functions without a thorough understanding of their operation.

The next example, `SQRT`, uses the library function `sqrt()` to calculate the square root of a number entered by the user.

```
// sqrt.cpp
// demonstrates sqrt() library function
#include <iostream>           //for cout, etc.
#include <cmath>              //for sqrt()
using namespace std;

int main()
{
    double number, answer;    //sqrt() requires type double

    cout << "Enter a number: ";
    cin >> number;            //get the number
    answer = sqrt(number);    //find square root
    cout << "Square root is "
         << answer << endl;  //display it
    return 0;
}
```



The program first obtains a number from the user. This number is then used as an argument to the `sqrt()` function, in the statement

```
answer = sqrt(number);
```

An *argument* is the input to the function; it is placed inside the parentheses following the function name. The function then processes the argument and returns a value; this is the output from the function. In this case the return value is the square root of the original number. Returning a value means that the function expression takes on this value, which can then be assigned to another variable—in this case `answer`. The program then displays this value. Here's some output from the program:

```
Enter a number: 1000  
Square root is 31.622777
```

Multiplying 31.622777 by itself on your pocket calculator will verify that this answer is pretty close.

The arguments to a function, and their return values, must be the correct data type. You can find what these data types are by looking at the description of the library function in your compiler's help file, which describes each of the hundreds of library functions. For `sqrt()`, the description specifies both an argument and a return value of type `double`, so we use variables of this type in the program.

## Header Files

As with `cout` and other such objects, you must `#include` a header file that contains the declaration of any library functions you use. In the documentation for the `sqrt()` function, you'll see that the specified header file is `CMATH`. In `SQRT` the preprocessor directive

```
#include <cmath>
```

takes care of incorporating this header file into our source file.

If you don't include the appropriate header file when you use a library function, you'll get an error message like this from the compiler: *'sqrt' unidentified identifier*.

## Library Files

We mentioned earlier that various files containing library functions and objects will be linked to your program to create an executable file. These files contain the actual machine-executable code for the functions. Such library files often have the extension `.LIB`. The `sqrt()` function is found in such a file. It is automatically extracted from the file by the linker, and the proper connections are made so that it can be called (that is, invoked or accessed) from the `SQRT` program. Your compiler takes care of all these details for you, so ordinarily you don't need to worry about the process. However, you should understand what these files are for.

## Header Files and Library Files

The relationship between library files and header files can be confusing, so let's review it. To use a library function like `sqrt()`, you must link the library file that contains it to your program. The appropriate functions from the library file are then connected to your program by the linker.

However, that's not the end of the story. The functions in your source file need to know the names and types of the functions and other elements in the library file. They are given this information in a header file. Each header file contains information for a particular group of functions. The functions themselves are grouped together in a library file, but the information about them is scattered throughout a number of header files. The `iostream` header file contains information for various I/O functions and objects, including `cout`, while the `cmath` header file contains information for mathematics functions such as `sqrt()`. If you were using string functions such as `strcpy()`, you would include `string.h`, and so on.

Figure 2.12 shows the relationship of header files and library files to the other files used in program development.

The use of header files is common in C++. Whenever you use a library function or a predefined object or operator, you will need to use a header file that contains appropriate declarations.

## Two Ways to Use `#include`

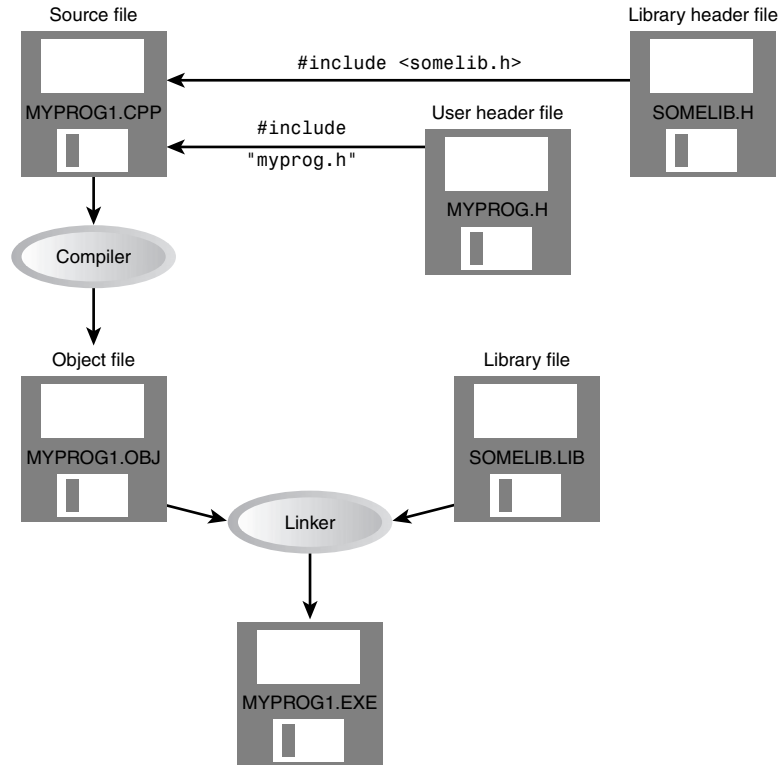
You can use `#include` in two ways. The angle brackets `<` and `>` surrounding the filenames `iostream` and `cmath` in the `SQRT` example indicate that the compiler should begin searching for these files in the standard `INCLUDE` directory. This directory, which is traditionally called `INCLUDE`, holds the header files supplied by the compiler manufacturer for the system.

Instead of angle brackets around the filename, you can also use quotation marks, as in

```
#include "myheader.h"
```

Quotation marks instruct the compiler to begin its search for the header file in the current directory; this is usually the directory that contains the source file. You normally use quotation marks for header files you write yourself (a situation we'll explore in Chapter 13, "Multifile Programs"). Quotation marks or angle brackets work in any case, but making the appropriate choice speeds up the compilation process slightly by giving the compiler a hint about where to find the file.

Appendix C, "Microsoft Visual C++," and Appendix D, "Borland C++Builder," explain how to handle header files with specific compilers.

**FIGURE 2.12**

*Header and library files.*

## Summary

In this chapter we've learned that a major building block of C++ programs is the *function*. A function named `main()` is always the first one executed when a program is executed.

A function is composed of *statements*, which tell the computer to do something. Each statement ends with a semicolon. A statement may contain one or more *expressions*, which are sequences of variables and operators that usually evaluate to a specific value.

Output is most commonly handled in C++ with the `cout` object and `<<` insertion operator, which together cause variables or constants to be sent to the standard output device—usually the screen. Input is handled with `cin` and the extraction operator `>>`, which cause values to be received from the standard input device—usually the keyboard.

Various data types are built into C++: `char`, `int`, `long`, and `short` are the integer types and `float`, `double`, and `long double` are the floating-point types. All of these types are signed. Unsigned versions of the integer types, signaled by the keyword `unsigned`, don't hold negative numbers but hold positive ones twice as large. Type `bool` is used for Boolean variables and can hold only the constants `true` or `false`.

The `const` keyword stipulates that a variable's value will not change in the course of a program. Strictly speaking, the variable is no longer a variable but a constant.

A variable is automatically converted from one type to another in mixed expressions (those involving different data types) and by casting, which allows the programmer to specify a conversion.

C++ employs the usual arithmetic operators `+`, `-`, `*`, and `/`. In addition, the remainder operator, `%`, returns the remainder of integer division.

The arithmetic assignment operators `+=`, `+-`, and so on perform an arithmetic operation and an assignment simultaneously. The increment and decrement operators `++` and `--` increase or decrease a variable by 1.

Preprocessor directives consist of instructions to the compiler, rather than to the computer. The `#include` directive tells the compiler to insert another file into the present source file, and the `#define` directive tells it to substitute one thing for another. The `using` directive tells the compiler to recognize names that are in a certain namespace.

If you use a library function in your program, the code for the function is in a library file, which is automatically linked to your program. A header file containing the function's declaration must be inserted into your source file with an `#include` statement.

## Questions

Answers to these questions can be found in Appendix G.

1. Dividing a program into functions
  - a. is the key to object-oriented programming.
  - b. makes the program easier to conceptualize.
  - c. may reduce the size of the program.
  - d. makes the program run faster.
2. A function name must be followed by \_\_\_\_\_.
3. A function body is delimited by \_\_\_\_\_.
4. Why is the `main()` function special?
5. A C++ instruction that tells the computer to do something is called a \_\_\_\_\_.

6. Write an example of a normal C++ comment and an example of an old-fashioned `/*` comment.
7. An expression
  - a. usually evaluates to a numerical value.
  - b. indicates the emotional state of the program.
  - c. always occurs outside a function.
  - d. may be part of a statement.
8. Specify how many bytes are occupied by the following data types in a 32-bit system:
  - a. Type `int`
  - b. Type `long double`
  - c. Type `float`
  - d. Type `long`
9. True or false: A variable of type `char` can hold the value 301.
10. What kind of program elements are the following?
  - a. 12
  - b. `'a'`
  - c. 4.28915
  - d. `JungleJim`
  - e. `JungleJim()`
11. Write statements that display on the screen
  - a. the character `'x'`
  - b. the name *Jim*
  - c. the number 509
12. True or false: In an assignment statement, the value on the left of the equal sign is always equal to the value on the right.
13. Write a statement that displays the variable `george` in a field 10 characters wide.
14. What header file must you `#include` with your source file to use `cout` and `cin`?
15. Write a statement that gets a numerical value from the keyboard and places it in the variable `temp`.
16. What header file must you `#include` with your program to use `setw`?
17. Two exceptions to the rule that the compiler ignores whitespace are \_\_\_\_\_ and \_\_\_\_\_.

18. True or false: It's perfectly all right to use variables of different data types in the same arithmetic expression.
19. The expression `11%3` evaluates to \_\_\_\_\_.
20. An arithmetic assignment operator combines the effect of what two operators?
21. Write a statement that uses an arithmetic assignment operator to increase the value of the variable `temp` by 23. Write the same statement without the arithmetic assignment operator.
22. The increment operator increases the value of a variable by how much?
23. Assuming `var1` starts with the value 20, what will the following code fragment print out?  

```
cout << var1--;  
cout << ++var1;
```
24. In the examples we've seen so far, header files have been used for what purpose?
25. The actual code for library functions is contained in a \_\_\_\_\_ file.

## Exercises

Answers to the starred exercises can be found in Appendix G.

- \*1. Assuming there are 7.481 gallons in a cubic foot, write a program that asks the user to enter a number of gallons, and then displays the equivalent in cubic feet.
- \*2. Write a program that generates the following table:

1990	135
1991	7290
1992	11300
1993	16200

Use a single `cout` statement for all output.

- \*3. Write a program that generates the following output:

```
10  
20  
19
```

Use an integer constant for the 10, an arithmetic assignment operator to generate the 20, and a decrement operator to generate the 19.

4. Write a program that displays your favorite poem. Use an appropriate escape sequence for the line breaks. If you don't have a favorite poem, you can borrow this one by Ogden Nash:

```
Candy is dandy,  
But liquor is quicker.
```

5. A library function, `islower()`, takes a single character (a letter) as an argument and returns a nonzero integer if the letter is lowercase, or zero if it is uppercase. This function requires the header file `CTYPE.H`. Write a program that allows the user to enter a letter, and then displays either zero or nonzero, depending on whether a lowercase or uppercase letter was entered. (See the `SQRT` program for clues.)
6. On a certain day the British pound was equivalent to \$1.487 U.S., the French franc was \$0.172, the German deutschemark was \$0.584, and the Japanese yen was \$0.00955. Write a program that allows the user to enter an amount in dollars, and then displays this value converted to these four other monetary units.
7. You can convert temperature from degrees Celsius to degrees Fahrenheit by multiplying by 9/5 and adding 32. Write a program that allows the user to enter a floating-point number representing degrees Celsius, and then displays the corresponding degrees Fahrenheit.
8. When a value is smaller than a field specified with `setw()`, the unused locations are, by default, filled in with spaces. The manipulator `setfill()` takes a single character as an argument and causes this character to be substituted for spaces in the empty parts of a field. Rewrite the `WIDTH` program so that the characters on each line between the location name and the population number are filled in with periods instead of spaces, as in

```
Portcity.....2425785
```

9. If you have two fractions,  $a/b$  and  $c/d$ , their sum can be obtained from the formula

$$\begin{array}{r} a \qquad c \qquad a*d + b*c \\ --- + --- = ----- \\ b \qquad d \qquad b*d \end{array}$$

For example, 1/4 plus 2/3 is

$$\begin{array}{r} 1 \qquad 2 \qquad 1*3 + 4*2 \qquad 3 + 8 \qquad 11 \\ --- + --- = ----- = ----- = ----- \\ 4 \qquad 3 \qquad 4*3 \qquad 12 \qquad 12 \end{array}$$

Write a program that encourages the user to enter two fractions, and then displays their sum in fractional form. (You don't need to reduce it to lowest terms.) The interaction with the user might look like this:

```
Enter first fraction: 1/2
    Enter second fraction: 2/5
    Sum = 9/10
```

You can take advantage of the fact that the extraction operator (`>>`) can be chained to read in more than one quantity at once:

```
cin >> a >> dummychar >> b;
```

10. In the heyday of the British empire, Great Britain used a monetary system based on pounds, shillings, and pence. There were 20 shillings to a pound, and 12 pence to a shilling. The notation for this old system used the pound sign, £, and two decimal points, so that, for example, £5.2.8 meant 5 pounds, 2 shillings, and 8 pence. (*Pence* is the plural of *penny*.) The new monetary system, introduced in the 1950s, consists of only pounds and pence, with 100 pence to a pound (like U.S. dollars and cents). We'll call this new system *decimal pounds*. Thus £5.2.8 in the old notation is £5.13 in decimal pounds (actually £5.1333333). Write a program to convert the old pounds-shillings-pence format to decimal pounds. An example of the user's interaction with the program would be

```
Enter pounds: 7
Enter shillings: 17
Enter pence: 9
Decimal pounds = £7.89
```

In most compilers you can use the decimal number 156 (hex character constant '\x9c') to represent the pound sign (£). In some compilers, you can put the pound sign into your program directly by pasting it from the Windows Character Map accessory.

11. By default, output is right-justified in its field. You can left-justify text output using the manipulator `setiosflags(ios::left)`. (For now, don't worry about what this new notation means.) Use this manipulator, along with `setw()`, to help generate the following output:

Last name	First name	Street address	Town	State
Jones	Bernard	109 Pine Lane	Littletown	MI
O'Brian	Coleen	42 E. 99th Ave.	Bigcity	NY
Wong	Harry	121-A Alabama St.	Lakeville	IL

12. Write the inverse of Exercise 10, so that the user enters an amount in Great Britain's new decimal-pounds notation (pounds and pence), and the program converts it to the old pounds-shillings-pence notation. An example of interaction with the program might be

```
Enter decimal pounds: 3.51
Equivalent in old notation = £3.10.2.
```

Make use of the fact that if you assign a floating-point value (say 12.34) to an integer variable, the decimal fraction (0.34) is lost; the integer value is simply 12. Use a cast to avoid a compiler warning. You can use statements like

```
float decpounds;    // input from user (new-style pounds)
int pounds;         // old-style (integer) pounds
float decfrac;      // decimal fraction (smaller than 1.0)
```

```
pounds = static_cast<int>(decpounds); // remove decimal fraction
decfrac = decpounds - pounds; // regain decimal fraction
```

You can then multiply `decfrac` by 20 to find shillings. A similar operation obtains pence.



