

**More than 450,000 Programmers
Have Learned C++ from Previous Editions**



C++ PRIMER

FOURTH EDITION



Stanley B. Lippman
Josée Lajoie
Barbara E. Moo

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

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

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

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

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

International Sales
international@pearsoned.com

Visit us on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Lippman, Stanley B.

C++ primer / Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. — 4th ed.

p. cm.

Includes index.

ISBN 0-201-72148-1 (pbk. : alk. paper)

1. C++ (Computer program language) I. Lajoie, Josée. II. Moo, Barbara E. III. Title.

QA76.73.C153L57697 2005

005.13'3—dc22

2004029301

Copyright © 2005 Objectwrite Inc., Josée Lajoie and Barbara E. Moo

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
One Lake Street
Upper Saddle River, NJ 07458

ISBN 0-201-72148-1

Text printed in the United States on recycled paper at Courier Stoughton in Stoughton, Massachusetts.
11th Printing January 2010

Preface

C++ Primer, Fourth Edition, provides a comprehensive introduction to the C++ language. As a primer, it provides a clear tutorial approach to the language, enhanced by numerous examples and other learning aids. Unlike most primers, it also provides a detailed description of the language, with particular emphasis on current and effective programming techniques.

Countless programmers have used previous editions of *C++ Primer* to learn C++. In that time C++ has matured greatly. Over the years, the focus of the language—and of C++ programmers—has grown beyond a concentration on run-time efficiency to focus on ways of making *programmers* more efficient. With the widespread availability of the standard library, it is possible to use and learn C++ more effectively than in the past. This revision of the *C++ Primer* reflects these new possibilities.

Changes to the Fourth Edition

In this edition, we have completely reorganized and rewritten the *C++ Primer* to highlight modern styles of C++ programming. This edition gives center stage to using the standard library while deemphasizing techniques for low-level programming. We introduce the standard library much earlier in the text and have reformulated the examples to take advantage of library facilities. We have also streamlined and reordered the presentation of language topics.

In addition to restructuring the text, we have incorporated several new elements to enhance the reader’s understanding. Each chapter concludes with a Chapter Summary and glossary of Defined Terms, which recap the chapter’s most important points. Readers should use these sections as a personal checklist: If you do not understand a term, restudy the corresponding part of the chapter.

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

- Important terms are indicated in **bold**; important terms that we assume are already familiar to the reader are indicated in ***bold italics***. Each term appears in the chapter’s Defined Terms section.
- Throughout the book, we highlight parts of the text to call attention to important aspects of the language, warn about common pitfalls, suggest good programming practices, and provide general usage tips. We hope that these notes will help readers more quickly digest important concepts and avoid common pitfalls.

- To make it easier to follow the relationships among features and concepts, we provide extensive forward and backward cross-references.
- We have provided sidebar discussions that focus on important concepts and supply additional explanations for topics that programmers new to C++ often find most difficult.
- Learning any programming language requires writing programs. To that end, the primer provides extensive examples throughout the text. Source code for the extended examples is available on the Web at the following URL:

http://www.awprofessional.com/cpp_primer

What hasn't changed from earlier versions is that the book remains a comprehensive tutorial introduction to C++. Our intent is to provide a clear, complete and correct guide to the language. We teach the language by presenting a series of examples, which, in addition to explaining language features, show how to make the best use of C++. Although knowledge of C (the language on which C++ was originally based) is not assumed, we do assume the reader has programmed in a modern block-structured language.

Structure of This Book

C++ Primer provides an introduction to the International Standard on C++, covering both the language proper and the extensive library that is part of that standard. Much of the power of C++ comes from its support for programming with abstractions. Learning to program effectively in C++ requires more than learning new syntax and semantics. Our focus is on how to use the features of C++ to write programs that are safe, that can be built quickly, and yet offer performance comparable to the sorts of low-level programs often written in C.

C++ is a large language and can be daunting to new users. Modern C++ can be thought of as comprising three parts:

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

Most texts present C++ in this same order: They start by covering the low-level details and then introduce the more advanced language features. They explain the standard library only after having covered the entire language. The result, all too often, is that readers get bogged down in issues of low-level programming or the complexities of writing type definitions and never really understand the power of programming in a more abstract way. Needless to say, readers also often do not learn enough to build their own abstractions.

In this edition we take a completely different tack. We start by covering the basics of the language and the library together. Doing so allows you, the reader, to

write significant programs. Only after a thorough grounding in using the library—and writing the kinds of abstract programs that the library allows—do we move on to those features of C++ that will enable you to write your own abstractions.

Parts I and II cover the basic language and library facilities. The focus of these parts is to learn how to write C++ programs and how to use the abstractions from the library. Most C++ programmers need to know essentially everything covered in this portion of the book.

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

Parts III through V focus on how we can write our own types. Part III introduces the heart of C++: its support for classes. The class mechanism provides the basis for writing our own abstractions. Classes are also the foundation for object-oriented and generic programming, which we cover in Part IV. The *Primer* concludes with Part V, which covers advanced features that are of most use in structuring large, complex systems.

Acknowledgments

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

We also extend our deep-felt thanks to our reviewers, whose helpful comments on multiple drafts led us to make improvements great and small throughout the book: Paul Abrahams, Michael Ball, Mary Dageforde, Paul DuBois, Matt Greenwood, Matthew P. Johnson, Andrew Koenig, Nevin Liber, Bill Locke, Robert Murray, Phil Romanik, Justin Shaw, Victor Shtern, Clovis Tondo, Daveed Vandevoorde, and Steve Vinoski.

This book was typeset using L^AT_EX and the many packages that accompany the L^AT_EX distribution. Our well-justified thanks go to the members of the L^AT_EX community, who have made available such powerful typesetting tools.

The examples in this book have been compiled on the GNU and Microsoft compilers. Our thanks to their developers, and to those who have developed all the other C++ compilers, thereby making C++ a reality.

Finally, we thank the fine folks at Addison-Wesley who have shepherded this edition through the publishing process: Debbie Lafferty, our original editor, who initiated this edition and who had been with the *Primer* from its very first edition; Peter Gordon, our new editor, whose insistence on updating and streamlining the text have, we hope, greatly improved the presentation; Kim Boedigheimer, who keeps us all on schedule; and Tyrrell Albaugh, Jim Markham, Elizabeth Ryan, and John Fuller, who saw us through the design and production process.

C H A P T E R 1

G E T T I N G S T A R T E D

CONTENTS

Section 1.1	Writing a Simple C++ Program	2
Section 1.2	A First Look at Input/Output	5
Section 1.3	A Word About Comments	10
Section 1.4	Control Structures	11
Section 1.5	Introducing Classes	20
Section 1.6	The C++ Program	25
Chapter Summary	28
Defined Terms	28

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

Having read this chapter and worked through the exercises, the reader should be able to write, compile, and execute simple programs. Subsequent chapters will explain in more detail the topics introduced here.

Learning a new programming language requires writing programs. In this chapter, we'll write a program to solve a simple problem that represents a common data-processing task: A bookstore keeps a file of transactions, each of which records the sale of a given book. Each transaction contains an ISBN (International Standard Book Number, a unique identifier assigned to most books published throughout the world), the number of copies sold, and the price at which each copy was sold. Each transaction looks like

0-201-70353-X 4 24.99

where the first element is the ISBN, the second is the number of books sold, and the last is the sales price. Periodically the bookstore owner reads this file and computes the number of copies of each title sold, the total revenue from that book, and the average sales price. We want to supply a program do these computations.

Before we can write this program we need to know some basic features of C++. At a minimum we'll need to know how to write, compile, and execute a simple program. What must this program do? Although we have not yet designed our solution, we know that the program must

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

We'll start by reviewing these parts of C++ and then write a solution to our bookstore problem.

1.1 Writing a Simple C++ Program

Every C++ program contains one or more *functions*, one of which must be named `main`. A function consists of a sequence of *statements* that perform the work of the function. The operating system executes a program by calling the function named `main`. That function executes its constituent statements and returns a value to the operating system.

Here is a simple version of `main` does nothing but return a value:

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

The operating system uses the value returned by `main` to determine whether the program succeeded or failed. A return value of 0 indicates success.

The `main` function is special in various ways, the most important of which are that the function must exist in every C++ program and it is the (only) function that the operating system explicitly calls.

We define `main` the same way we define other functions. A function definition specifies four elements: the *return type*, the *function name*, a (possibly empty) *parameter list* enclosed in parentheses, and the *function body*. The `main` function may have only a restricted set of parameters. As defined here, the parameter list is empty; Section 7.2.6 (p. 243) will cover the other parameters that can be defined for `main`.

The `main` function is required to have a return type of `int`, which is the type that represents integers. The `int` type is a **built-in type**, which means that the type is defined by the language.

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

```
{  
    return 0;  
}
```

The only statement in our program is a `return`, which is a statement that terminates a function.



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

When the `return` includes a value such as `0`, that value is the return value of the function. The value returned must have the same type as the return type of the function or be a type that can be converted to that type. In the case of `main` the return type must be `int`, and the value `0` is an `int`.

On most systems, the return value from `main` is a status indicator. A return value of `0` indicates the successful completion of `main`. Any other return value has a meaning that is defined by the operating system. Usually a nonzero return indicates that an error occurred. Each operating system has its own way of telling the user what `main` returned.

1.1.1 Compiling and Executing Our Program

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

Many PC-based compilers are run from an integrated development environment (IDE) that bundles the compiler with associated build and analysis tools. These environments can be a great asset in developing complex programs but require a fair bit of time to learn how to use effectively. Most of these environments include a point-and-click interface that allows the programmer to write a program and use various menus to compile and execute the program. Learning how to use such environments is well beyond the scope of this book.

Most compilers, including those that come with an IDE, provide a command-line interface. Unless you are already familiar with using your compiler's IDE,

it can be easier to start by using the simpler, command-line interface. Using the command-line interface lets you avoid the overhead of learning the IDE before learning the language.

Program Source File Naming Convention

Whether we are using a command-line interface or an IDE, most compilers expect that the program we want to compile will be stored in a file. Program files are referred to as **source files**. On most systems, a source file has a name that consists of two parts: a file name—for example, `prog1`—and a file suffix. By convention, the suffix indicates that the file is a program. The suffix often also indicates what language the program is written in and selects which compiler to run. The system that we used to compile the examples in this book treats a file with a suffix of `.cc` as a C++ program and so we stored this program as

```
prog1.cc
```

The suffix for C++ program files depends on which compiler you're running. Other conventions include

```
prog1.cxx  
prog1.cpp  
prog1.cp  
prog1.C
```

INVOKING THE GNU OR MICROSOFT COMPILERS

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

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

where `$` is the system prompt. This command generates an executable file named `prog1` or `prog1.exe`, depending on the operating system. On UNIX, executable files have no suffix; on Windows, the suffix is `.exe`. The `-o prog1` is an argument to the compiler and names the file in which to put the executable file. If the `-o prog1` is omitted, then the compiler generates an executable named `a.out` on UNIX systems and `a.exe` on Windows.

The Microsoft compilers are invoked using the command `c1`:

```
C:\directory> c1 -GX prog1.cpp
```

where `C:\directory>` is the system prompt and `directory` is the name of the current directory. The command to invoke the compiler is `c1`, and `-GX` is an option that is required for programs compiled using the command-line interface. The Microsoft compiler automatically generates an executable with a name that corresponds to the source file name. The executable has the suffix `.exe` and the same name as the source file name. In this case, the executable is named `prog1.exe`.

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

Running the Compiler from the Command Line

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

```
$ CC prog1.cc
```

where `CC` names the compiler and `$` represents the system prompt. The output of the compiler is an executable file that we invoke by naming it. On our system, the compiler generates the executable in a file named `a.exe`. UNIX compilers tend to put their executables in a file named `a.out`. To run an executable we supply that name at the command-line prompt:

```
$ a.exe
```

executes the program we compiled. On UNIX systems you sometimes must also specify which directory the file is in, even if it is in the current directory. In such cases, we would write

```
$ ./a.exe
```

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

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

```
$ echo $?
```

To see the status on a Windows system, we write

```
C:\directory> echo %ERRORLEVEL%
```

EXERCISES SECTION 1.1.1

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

Exercise 1.2: Change the program to return `-1`. A return value of `-1` is often treated as an indicator that the program failed. However, systems vary as to how (or even whether) they report a failure from `main`. Recompile and rerun your program to see how your system treats a failure indicator from `main`.

1.2 A First Look at Input/Output

C++ does not directly define any statements to do input or output (IO). Instead, IO is provided by the **standard library**. The IO library provides an extensive set of

facilities. However, for many purposes, including the examples in this book, one needs to know only a few basic concepts and operations.

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

1.2.1 Standard Input and Output Objects

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

Ordinarily, the system associates each of these objects with the window in which the program is executed. So, when we read from **cin**, data is read from the window in which the program is executing, and when we write to **cout**, **cerr**, or **clog**, the output is written to the same window. Most operating systems give us a way of redirecting the input or output streams when we run a program. Using redirection we can associate these streams with files of our choosing.

1.2.2 A Program that Uses the IO Library

So far, we have seen how to compile and execute a simple program, although that program did no work. In our overall problem, we’ll have several records that refer to the same ISBN. We’ll need to consolidate those records into a single total, implying that we’ll need to know how to add the quantities of books sold.

To see how to solve part of that problem, let’s start by looking at how we might add two numbers. Using the IO library, we can extend our **main** program to ask the user to give us two numbers and then print their sum:

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

This program starts by printing

```
Enter two numbers:
```

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

```
3 7
```

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

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

The first line of our program is a **preprocessor directive**:

```
#include <iostream>
```

which tells the compiler that we want to use the `iostream` library. The name inside angle brackets is a **header**. Every program that uses a library facility must include its associated header. The `#include` directive must be written on a single line—the name of the header and the `#include` must appear on the same line. In general, `#include` directives should appear outside any function. Typically, all the `#include` directives for a program appear at the beginning of the file.

Writing to a Stream

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

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

This statement uses the output operator twice. Each instance of the output operator takes two operands: The left-hand operand must be an `ostream` object; the right-hand operand is a value to print. The operator writes its right-hand operand to the `ostream` that is its left-hand operand.

In C++ every expression produces a result, which typically is the value generated by applying an operator to its operands. In the case of the output operator, the result is the value of its left-hand operand. That is, the value returned by an output operation is the output stream itself.

The fact that the operator returns its left-hand operand allows us to chain together output requests. The statement that prints our prompt is equivalent to

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

Because `(std::cout << "Enter two numbers:")` returns its left operand, `std::cout`, this statement is equivalent to

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

`endl` is a special value, called a **manipulator**, that when written to an output stream has the effect of writing a newline to the output and flushing the *buffer* associated with that device. By flushing the buffer, we ensure that the user will see the output written to the stream immediately.



Programmers often insert print statements during debugging. Such statements should always flush the stream. Forgetting to do so may cause output to be left in the buffer if the program crashes, leading to incorrect inferences about where the program crashed.

Using Names from the Standard Library

Careful readers will note that this program uses `std::cout` and `std::endl` rather than just `cout` and `endl`. The prefix `std::` indicates that the names `cout` and `endl` are defined inside the **namespace** named `std`. Namespaces allow programmers to avoid inadvertent collisions with the same names defined by a library. Because the names that the standard library defines are defined in a namespace, we can use the same names for our own purposes.

One side effect of the library's use of a namespace is that when we use a name from the library, we must say explicitly that we want to use the name from the `std` namespace. Writing `std::cout` uses the **scope operator** (the `::` operator) to say that we want to use the name `cout` that is defined in the namespace `std`. We'll see in Section 3.1 (p. 78) a way that programs often use to avoid this verbose syntax.

Reading From a Stream

Having written our prompt, we next want to read what the user writes. We start by defining two **variables** named `v1` and `v2` to hold the input:

```
int v1, v2;
```

We define these variables as type `int`, which is the built-in type representing integral values. These variables are **uninitialized**, meaning that we gave them no initial value. Our first use of these variables will be to read a value into them, so the fact that they have no initial value is okay.

The next statement

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

reads the input. The **input operator** (the `>>` operator) behaves analogously to the output operator. It takes an `istream` as its left-hand operand and an object as its right-hand operand. It reads from its `istream` operand and stores the value it read in its right-hand operand. Like the output operator, the input operator returns its left-hand operand as its result. Because the operator returns its left-hand operand, we can combine a sequence of input requests into a single statement. In other words, this input operation is equivalent to

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

The effect of our input operation is to read two values from the standard input, storing the first in `v1` and the second in `v2`.

Completing the Program

What remains is to print our result:

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

This statement, although it is longer than the statement that printed the prompt, is conceptually no different. It prints each of its operands to the standard output. What is interesting is that the operands are not all the same kinds of values. Some operands are **string literals**, such as

```
"The sum of "
```

and others are various `int` values, such as `v1`, `v2`, and the result of evaluating the arithmetic expression:

```
v1 + v2
```

The `iostream` library defines versions of the input and output operators that accept all of the built-in types.



When writing a C++ program, in most places that a space appears we could instead use a newline. One exception to this rule is that spaces inside a string literal cannot be replaced by a newline. Another exception is that spaces are not allowed inside preprocessor directives.

KEY CONCEPT: INITIALIZED AND UNINITIALIZED VARIABLES

Initialization is an important concept in C++ and one to which we will return throughout this book.

Initialized variables are those that are given a value when they are defined. Uninitialized variables are not given an initial value:

```
int val1 = 0;      // initialized  
int val2;         // uninitialized
```

It is almost always right to give a variable an initial value, but we are not required to do so. When we are certain that the first use of a variable gives it a new value, then there is no need to invent an initial value. For example, our first nontrivial program on page 6 defined uninitialized variables into which we immediately read values.

When we define a variable, we should give it an initial value unless we are *certain* that the initial value will be overwritten before the variable is used for any other purpose. If we cannot guarantee that the variable will be reset before being read, we should initialize it.

EXERCISES SECTION 1.2.2

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

Exercise 1.4: Our program used the built-in addition operator, `+`, to generate the sum of two numbers. Write a program that uses the multiplication operator, `*`, to generate the product of two numbers.

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

Exercise 1.6: Explain what the following program fragment does:

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

Is this code legal? If so, why? If not, why not?

1.3 A Word About Comments

Before our programs get much more complicated, we should see how C++ handles *comments*. Comments help the human readers of our programs. They are typically used to summarize an algorithm, identify the purpose of a variable, or clarify an otherwise obscure segment of code. Comments do not increase the size of the executable program. The compiler ignores all comments.



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

There are two kinds of comments in C++: single-line and paired. A single-line comment starts with a double slash (`//`). Everything to the right of the slashes on the current line is a comment and ignored by the compiler.

The other delimiter, the comment pair (`/* */`), is inherited from the C language. Such comments begin with a `/*` and end with the next `*/`. The compiler treats everything that falls between the `/*` and `*/` as part of the comment:

```
#include <iostream>  
/* Simple main function: Read two numbers and write their sum */  
int main()  
{  
    // prompt user to enter two numbers  
    std::cout << "Enter two numbers:" << std::endl;  
    int v1, v2;           // uninitialized  
    std::cin >> v1 >> v2; // read input  
    return 0;  
}
```

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

Programs typically contain a mixture of both comment forms. Comment pairs generally are used for multi-line explanations, whereas double slash comments tend to be used for half-line and single-line remarks.

Too many comments intermixed with the program code can obscure the code. It is usually best to place a comment block above the code it explains.

Comments should be kept up to date as the code itself changes. Programmers expect comments to remain accurate and so believe them, even when other forms of system documentation are known to be out of date. An incorrect comment is worse than no comment at all because it may mislead a subsequent reader.

Comment Pairs Do Not Nest

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

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

When commenting out a large section of a program, it can seem easiest to put a comment pair around a region that you want to omit temporarily. The trouble is that if that code already has a comment pair, then the newly inserted comment will terminate prematurely. A better way to temporarily ignore a section of code is to use your editor to insert single-line comment at the beginning of each line of code you want to ignore. That way, you need not worry about whether the code you are commenting out already contains a comment pair.

1.4 Control Structures

Statements execute sequentially: The first statement in a function is executed first, followed by the second, and so on. Of course, few programs—including the one we'll need to write to solve our bookstore problem—can be written using only sequential execution. Instead, programming languages provide various control

EXERCISES SECTION 1.3

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

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

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

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

structures that allow for more complicated execution paths. This section will take a brief look at some of the control structures provided by C++. Chapter 6 covers statements in detail.

1.4.1 The `while` Statement

A **while statement** provides for iterative execution. We could use a while to write a program to sum the numbers from 1 through 10 inclusive as follows:

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

This program when compiled and executed will print:

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

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

The important part is the `while` statement. A `while` has the form

```
while (condition) while_body_statement;
```

A `while` executes by (repeatedly) testing the `condition` and executing the associated `while_body_statement` until the `condition` is false.

A **condition** is an expression that is evaluated so that its result can be tested. If the resulting value is nonzero, then the condition is true; if the value is zero then the condition is false.

If the *condition* is true (the expression evaluates to a value other than zero) then *while_body_statement* is executed. After executing *while_body_statement*, the *condition* is tested again. If *condition* remains true, then the *while_body_statement* is again executed. The while continues, alternatively testing the *condition* and executing *while_body_statement* until the *condition* is false.

In this program, the while statement is:

```
// keep executing the while until val is greater than 10
while (val <= 10) {
    sum += val; // assigns sum + val to sum
    ++val;       // add 1 to val
}
```

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

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

A block is a sequence of statements enclosed by curly braces. In C++, a block may be used wherever a statement is expected. The first statement in the block uses the **compound assignment operator**, (the `+=` operator). This operator adds its right-hand operand to its left-hand operand. It has the same effect as writing an addition and an **assignment**:

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

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

The next statement

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

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

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

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

KEY CONCEPT: INDENTATION AND FORMATTING OF C++ PROGRAMS

C++ programs are largely free-format, meaning that the positioning of curly braces, indentation, comments, and newlines usually has no effect on the meaning of our programs. For example, the curly brace that denotes the beginning of the body of `main` could be on the same line as `main`, positioned as we have done, at the beginning of the next line, or placed anywhere we'd like. The only requirement is that it be the first nonblank, noncomment character that the compiler sees after the close parenthesis that concludes `main`'s parameter list.

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

Endless debates occur as to the right way to format C or C++ programs. Our belief is that there is no single correct style but that there is value in consistency. We tend to put the curly braces that delimit functions on their own lines. We tend to indent compound input or output expressions so that the operators line up, as we did with the statement that wrote the output in the `main` function on page 6. Other indentation conventions will become clear as our programs become more complex.

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

1.4.2 The `for` Statement

In our `while` loop, we used the variable `val` to control how many times we iterated through the loop. On each pass through the `while`, the value of `val` was tested and then in the body the value of `val` was incremented.

The use of a variable like `val` to control a loop happens so often that the language defines a second control structure, called a **for statement**, that abbreviates the code that manages the loop variable. We could rewrite the program to sum the numbers from 1 through 10 using a `for` loop as follows:

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 up to 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum = sum + val

    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

Prior to the `for` loop, we define `sum`, which we set to zero. The variable `val` is used only inside the iteration and is defined as part of the `for` statement itself. The `for` statement

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

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

```
int val = 1;
```

defines an `int` object named `val` and gives it an initial value of one. The *init-statement* is performed only once, on entry to the `for`. The *condition*

```
val <= 10
```

which compares the current value in `val` to 10, is tested each time through the loop. As long as `val` is less than or equal to 10, we execute the `for` body. Only after executing the body is the *expression* executed. In this `for`, the expression uses the prefix increment operator, which as we know adds one to the value of `val`. After executing the *expression*, the `for` retests the *condition*. If the new value of `val` is still less than or equal to 10, then the `for` loop body is executed and `val` is incremented again. Execution continues until the *condition* fails.

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

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

The body uses the compound assignment operator to add the current value of `val` to `sum`, storing the result back into `sum`.

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

1. Create `val` and initialize it to 1.
2. Test whether `val` is less than or equal to 10.
3. If `val` is less than or equal to 10, execute the `for` body, which adds `val` to `sum`. If `val` is not less than or equal to 10, then break out of the loop and continue execution with the first statement following the `for` body.
4. Increment `val`.
5. Repeat the test in step 2, continuing with the remaining steps as long as the condition is true.



When we exit the `for` loop, the variable `val` is no longer accessible. It is not possible to use `val` after this loop terminates. However, not all compilers enforce this requirement.

In pre-Standard C++ names defined in a `for` header *were* accessible outside the `for` itself. This change in the language definition can surprise people accustomed to using an older compiler when they instead use a compiler that adheres to the standard.

COMPILATION REVISITED

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

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

```
//  error: missing ')' in parameter list for main
int main ( {
    //  error: used colon, not a semicolon after endl
    std::cout << "Read each file." << std::endl;
    //  error: missing quotes around string literal
    std::cout << Update master. << std::endl;
    //  ok: no errors on this line
    std::cout << "Write new master." << std::endl;
    //  error: missing ';' on return statement
    return 0
}
```

2. **Type errors.** Each item of data in C++ has an associated type. The value 10, for example, is an integer. The word "hello" surrounded by double quotation marks is a string literal. One example of a type error is passing a string literal to a function that expects an integer argument.
3. **Declaration errors.** Every name used in a C++ program must be declared before it is used. Failure to declare a name usually results in an error message. The two most common declaration errors are to forget to use `std::` when accessing a name from the library or to inadvertently misspell the name of an identifier:

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

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

EXERCISES SECTION 1.4.2

Exercise 1.9: What does the following `for` loop do? What is the final value of `sum`?

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

Exercise 1.10: Write a program that uses a `for` loop to sum the numbers from 50 to 100. Now rewrite the program using a `while`.

Exercise 1.11: Write a program using a `while` loop to print the numbers from 10 down to 0. Now rewrite the program using a `for`.

Exercise 1.12: Compare and contrast the loops you wrote in the previous two exercises. Are there advantages or disadvantages to using either form?

Exercise 1.13: Compilers vary as to how easy it is to understand their diagnostics. Write programs that contain the common errors discussed in the box on 16. Study the messages the compiler generates so that these messages will be familiar when you encounter them while compiling more complex programs.

1.4.3 The `if` Statement

A logical extension of summing the values between 1 and 10 is to sum the values between two numbers our user supplies. We might use the numbers directly in our `for` loop, using the first input as the lower bound for the range and the second as the upper bound. However, if the user gives us the higher number first, that strategy would fail: Our program would exit the `for` loop immediately. Instead, we should adjust the range so that the larger number is the upper bound and the smaller is the lower. To do so, we need a way to see which number is larger.

Like most languages, C++ provides an **`if` statement** that supports conditional execution. We can use an `if` to write our revised sum program:

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // read input
    // use smaller number as lower bound for summation
    // and larger number as upper bound
    int lower, upper;
    if (v1 <= v2) {
        lower = v1;
        upper = v2;
    } else {
        lower = v2;
        upper = v1;
    }
```

```

int sum = 0;
// sum values from lower up to and including upper
for (int val = lower; val <= upper; ++val)
    sum += val; // sum = sum + val

std::cout << "Sum of " << lower
           << " to " << upper
           << " inclusive is "
           << sum << std::endl;

return 0;
}

```

If we compile and execute this program and give it as input the numbers 7 and 3, then the output of our program will be

```
Sum of 3 to 7 inclusive is 25
```

Most of the code in this program should already be familiar from our earlier examples. The program starts by writing a prompt to the user and defines four `int` variables. It then reads from the standard input into `v1` and `v2`. The only new code is the `if` statement

```

// use smaller number as lower bound for summation
// and larger number as upper bound
int lower, upper;
if (v1 <= v2) {
    lower = v1;
    upper = v2;
} else {
    lower = v2;
    upper = v1;
}

```

The effect of this code is to set `upper` and `lower` appropriately. The `if` condition tests whether `v1` is less than or equal to `v2`. If so, we perform the block that immediately follows the condition. This block contains two statements, each of which does an assignment. The first statement assigns `v1` to `lower` and the second assigns `v2` to `upper`.

If the condition is false—that is, if `v1` is larger than `v2`—then we execute the statement following the `else`. Again, this statement is a block consisting of two assignments. We assign `v2` to `lower` and `v1` to `upper`.

1.4.4 Reading an Unknown Number of Inputs

Another change we might make to our summation program on page 12 would be to allow the user to specify a set of numbers to sum. In this case we can't know how many numbers we'll be asked to add. Instead, we want to keep reading numbers until the program reaches the end of the input. When the input is finished, the program writes the total to the standard output:

EXERCISES SECTION 1.4.3

Exercise 1.14: What happens in the program presented in this section if the input values are equal?

Exercise 1.15: Compile and run the program from this section with two equal values as input. Compare the output to what you predicted in the previous exercise. Explain any discrepancy between what happened and what you predicted.

Exercise 1.16: Write a program to print the larger of two inputs supplied by the user.

Exercise 1.17: Write a program to ask the user to enter a series of numbers. Print a message saying how many of the numbers are negative numbers.

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

If we give this program the input

3 4 5 6

then our output will be

Sum is: 18

As usual, we begin by including the necessary headers. The first line inside `main` defines two `int` variables, named `sum` and `value`. We'll use `value` to hold each number we read, which we do inside the condition in the `while`:

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

What happens here is that to evaluate the condition, the input operation

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

is executed, which has the effect of reading the next number from the standard input, storing what was read in `value`. The input operator (Section 1.2.2, p. 8) returns its left operand. The condition tests that result, meaning it tests `std::cin`.

When we use an `istream` as a condition, the effect is to test the state of the stream. If the stream is valid—that is, if it is still possible to read another input—then the test succeeds. An `istream` becomes invalid when we hit *end-of-file* or encounter an invalid input, such as reading a value that is not an integer. An `istream` that is in an invalid state will cause the condition to fail.

Until we do encounter end-of-file (or some other input error), the test will succeed and we'll execute the body of the `while`. That body is a single statement that uses the compound assignment operator. This operator adds its right-hand operand into the left hand operand.

ENTERING AN END-OF-FILE FROM THE KEYBOARD

Operating systems use different values for end-of-file. On Windows systems we enter an end-of-file by typing a control-z—simultaneously type the “ctrl” key and a “z.” On UNIX systems, including Mac OS-X machines, it is usually control-d.

Once the test fails, the `while` terminates and we fall through and execute the statement following the `while`. That statement prints `sum` followed by `endl`, which prints a newline and flushes the buffer associated with `cout`. Finally, we execute the `return`, which as usual returns zero to indicate success.

EXERCISES SECTION 1.4.4

Exercise 1.18: Write a program that prompts the user for two numbers and writes each number in the range specified by the two numbers to the standard output.

Exercise 1.19: What happens if you give the numbers 1000 and 2000 to the program written for the previous exercise? Revise the program so that it never prints more than 10 numbers per line.

Exercise 1.20: Write a program to sum the numbers in a user-specified range, omitting the `if` test that sets the upper and lower bounds. Predict what happens if the input is the numbers 7 and 3, in that order. Now run the program giving it the numbers 7 and 3, and see if the results match your expectation. If not, restudy the discussion on the `for` and `while` loop until you understand what happened.

1.5 Introducing Classes

The only remaining feature we need to understand before solving our bookstore problem is how to write a *data structure* to represent our transaction data. In C++ we define our own data structure by defining a **class**. The class mechanism is one of the most important features in C++. In fact, a primary focus of the design of C++ is to make it possible to define **class types** that behave as naturally as the built-in types themselves. The library types that we've seen already, such as `istream` and `ostream`, are all defined as classes—that is, they are not strictly speaking part of the language.

Complete understanding of the class mechanism requires mastering a lot of information. Fortunately, it is possible to use a class that someone else has written without knowing how to define a class ourselves. In this section, we'll describe a simple class that we can use in solving our bookstore problem. We'll implement

this class in the subsequent chapters as we learn more about types, expressions, statements, and functions—all of which are used in defining classes.

To use a class we need to know three things:

1. What is its name?
2. Where is it defined?
3. What operations does it support?

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

1.5.1 The `Sales_item` Class

The purpose of the `Sales_item` class is to store an ISBN and keep track of the number of copies sold, the revenue, and average sales price for that book. How these data are stored or computed is not our concern. To use a class, we need not know anything about how it is implemented. Instead, what we need to know is what operations the class provides.

As we've seen, when we use library facilities such as IO, we must include the associated headers. Similarly, for our own classes, we must make the definitions associated with the class available to the compiler. We do so in much the same way. Typically, we put the class definition into a file. Any program that wants to use our class must include that file.

Conventionally, class types are stored in a file with a name that, like the name of a program source file, has two parts: a file name and a file suffix. Usually the file name is the same as the class defined in the header. The suffix usually is `.h`, but some programmers use `.H`, `.hpp`, or `.hxx`. Compilers usually aren't picky about header file names, but IDEs sometimes are. We'll assume that our class is defined in a file named `Sales_item.h`.

Operations on `Sales_item` Objects

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

```
Sales_item item;
```

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

In addition to being able to define variables of type `Sales_item`, we can perform the following operations on `Sales_item` objects:

- Use the addition operator, `+`, to add two `Sales_items`
- Use the input operator, `>>`, to read a `Sales_item` object

- Use the output operator, `<<`, to write a `Sales_item` object
- Use the assignment operator, `=`, to assign one `Sales_item` object to another
- Call the `same_isbn` function to determine if two `Sales_items` refer to the same book

Reading and Writing `Sales_items`

Now that we know the operations that the class provides, we can write some simple programs to use this class. For example, the following program reads data from the standard input, uses that data to build a `Sales_item` object, and writes that `Sales_item` object back onto the standard output:

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

If the input to this program is

0-201-70353-X 4 24.99

then the output will be

0-201-70353-X 4 99.96 24.99

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

This program starts with two `#include` directives, one of which uses a new form. The `iostream` header is defined by the standard library; the `Sales_item` header is not. `Sales_item` is a type that we ourselves have defined. When we use our own headers, we use quotation marks (" ") to surround the header name.



Headers for the standard library are enclosed in angle brackets (< >). Nonstandard headers are enclosed in double quotes (" ").

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

KEY CONCEPT: CLASSES DEFINE BEHAVIOR

As we go through these programs that use `Sales_item`, the important thing to keep in mind is that the author of the `Sales_item` class defined *all* the actions that can be performed by objects of this class. That is, the author of the `Sales_item` data structure defines what happens when a `Sales_item` object is created and what happens when the addition or the input and output operators are applied to `Sales_item` objects, and so on.

In general, only the operations defined by a class can be used on objects of the class type. For now, the only operations we know we can perform on `Sales_item` objects are the ones listed on page 21.

We'll see how these operations are defined in Sections 7.7.3 and 14.2.

Adding `Sales_items`

A slightly more interesting example adds two `Sales_item` objects:

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

If we give this program the following input

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

our output is

```
0-201-78345-X 5 110.22
```

This program starts by including the `Sales_item` and `iostream` headers. Next we define two `Sales_item` objects to hold the two transactions that we wish to sum. The output expression does the addition and prints the result. We know from the list of operations on page 21 that adding two `Sales_items` together creates a new object whose ISBN is that of its operands and whose number sold and revenue reflect the sum of the corresponding values in its operands. We also know that the items we add must represent the same ISBN.

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

EXERCISES SECTION 1.5.1

Exercise 1.21: The Web site (http://www.awprofessional.com/cpp_primer) contains a copy of `Sales_item.h` in the Chapter 1 code directory. Copy that file to your working directory. Write a program that loops through a set of book sales transactions, reading each transaction and writing that transaction to the standard output.

Exercise 1.22: Write a program that reads two `Sales_item` objects that have the same ISBN and produces their sum.

Exercise 1.23: Write a program that reads several transactions for the same ISBN. Write the sum of all the transactions that were read.

1.5.2 A First Look at Member Functions

Unfortunately, there is a problem with the program that adds `Sales_items`. What should happen if the input referred to two different ISBNs? It doesn't make sense to add the data for two different ISBNs together. To solve this problem, we'll first check whether the `Sales_item` operands refer to the same ISBNs:

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

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

The difference between this program and the previous one is the `if` test and its associated `else` branch. Before explaining the `if` condition, we know that what this program does depends on the condition in the `if`. If the test succeeds, then we write the same output as the previous program and return 0 indicating success. If the test fails, we execute the block following the `else`, which prints a message and returns an error indicator.

What Is a Member Function?

The `if` condition

```
// first check that item1 and item2 represent the same book
if (item1.same_isbn(item2)) {
```

calls a **member function** of the `Sales_item` object named `item1`. A member function is a function that is defined by a class. Member functions are sometimes referred to as the **methods** of the class.

Member functions are defined once for the class but are treated as members of each object. We refer to these operations as member functions because they (usually) operate on a specific object. In this sense, they are members of the object, even though a single definition is shared by all objects of the same type.

When we call a member function, we (usually) specify the object on which the function will operate. This syntax uses the **dot operator** (the “.” operator):

```
item1.same_isbn
```

means “the `same_isbn` member of the object named `item1`.” The dot operator fetches its right-hand operand from its left. The dot operator applies only to objects of class type: The left-hand operand must be an object of class type; the right-hand operand must name a member of that type.



Unlike most other operators, the right operand of the dot (“.”) operator is not an object or value; it is the name of a member.

When we use a member function as the right-hand operand of the dot operator, we usually do so to call that function. We execute a member function in much the same way as we do any function: To call a function, we follow the function name by the **call operator** (the “()” operator). The call operator is a pair of parentheses that encloses a (possibly empty) list of *arguments* that we pass to the function.

The `same_isbn` function takes a single argument, and that argument is another `Sales_item` object. The call

```
item1.same_isbn(item2)
```

passes `item2` as an argument to the function named `same_isbn` that is a member of the object named `item1`. This function compares the ISBN part of its argument, `item2`, to the ISBN in `item1`, the object on which `same_isbn` is called. Thus, the effect is to test whether the two objects refer to the same ISBN.

If the objects refer to the same ISBN, we execute the statement following the `if`, which prints the result of adding the two `Sales_item` objects together. Otherwise, if they refer to different ISBNs, we execute the `else` branch, which is a block of statements. The block prints an appropriate error message and exits the program, returning `-1`. Recall that the return from `main` is treated as a status indicator. In this case, we return a nonzero value to indicate that the program failed to produce the expected result.

1.6 The C++ Program

Now we are ready to solve our original bookstore problem: We need to read a file of sales transactions and produce a report that shows for each book the total revenue, average sales price, and the number of copies sold.

EXERCISES SECTION 1.5.2

Exercise 1.24: Write a program that reads several transactions. For each new transaction that you read, determine if it is the same ISBN as the previous transaction, keeping a count of how many transactions there are for each ISBN. Test the program by giving multiple transactions. These transactions should represent multiple ISBNs but the records for each ISBN should be grouped together.

We'll assume that all of the transactions for a given ISBN appear together. Our program will combine the data for each ISBN in a `Sales_item` object named `total`. Each transaction we read from the standard input will be stored in a second `Sales_item` object named `trans`. Each time we read a new transaction we'll compare it to the `Sales_item` object in `total`. If the objects refer to the same ISBN, we'll update `total`. Otherwise we'll print the value in `total` and reset it using the transaction we just read.

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

int main()
{
    // declare variables to hold running sum and data for the next record
    Sales_item total, trans;
    // is there data to process?
    if (std::cin >> total) {
        // if so, read the transaction records
        while (std::cin >> trans)
            if (total.same_isbn(trans))
                // match: update the running total
                total = total + trans;
            else {
                // no match: print & assign to total
                std::cout << total << std::endl;
                total = trans;
            }
        // remember to print last record
        std::cout << total << std::endl;
    } else {
        // no input!, warn the user
        std::cout << "No data?!" << std::endl;
        return -1; // indicate failure
    }
    return 0;
}
```

This program is the most complicated one we've seen so far, but it uses only facilities that we have already encountered. As usual, we begin by including the headers that we use: `iostream` from the library and `Sales_item.h`, which is our own header.

Inside `main` we define the objects we need: `total`, which we'll use to sum the data for a given ISBN, and `trans`, which will hold our transactions as we read them. We start by reading a transaction into `total` and testing whether the read was successful. If the read fails, then there are no records and we fall through to the outermost `else` branch, which prints a message to warn the user that there was no input.

Assuming we have successfully read a record, we execute the code in the `if` branch. The first statement is a `while` that will loop through all the remaining records. Just as we did in the program on page 18, our `while` condition reads a value from the standard input and then tests that valid data was actually read. In this case, we read a `Sales_item` object into `trans`. As long as the read succeeds, we execute the body of the `while`.

The body of the `while` is a single `if` statement. We test whether the ISBNs are equal, and if so we add the two objects and store the result in `total`. If the ISBNs are not equal, we print the value stored in `total` and reset `total` by assigning `trans` to it. After execution of the `if`, we return to the condition in the `while`, reading the next transaction and so on until we run out of records.

Once the `while` completes, we still must write the data associated with the last ISBN. When the `while` terminates, `total` contains the data for the last ISBN in the file, but we had no chance to print it. We do so in the last statement of the block that concludes the outermost `if` statement.

EXERCISES SECTION 1.6

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

Exercise 1.26: In the bookstore program we used the addition operator and not the compound assignment operator to add `trans` to `total`. Why didn't we use the compound assignment operator?

CHAPTER SUMMARY

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

DEFINED TERMS

argument A value passed to a function when it is called.

block Sequence of statements enclosed in curly braces.

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

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

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

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

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

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

clog `ostream` object tied to the standard error. By default, writes to `clog` are

buffered. Usually used to report information about program execution to a log file.

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

condition An expression that is evaluated as true or false. An arithmetic expression that evaluates to zero is false; any other value yields true.

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

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

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

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

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

expression The smallest unit of computation. An expression consists of one or more operands and usually an operator. Expressions are evaluated to produce a result. For example, assuming `i` and `j` are `ints`, then `i + j` is an arithmetic addition expression

and yields the sum of the two `int` values. Expressions are covered in more detail in Chapter 5.

for statement Control statement that provides iterative execution. Often used to step through a data structure or to repeat a calculation a fixed number of times.

function A named unit of computation.

function body Statement block that defines the actions performed by a function.

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

header A mechanism whereby the definitions of a class or other names may be made available to multiple programs. A header is included in a program through a `#include` directive.

if statement Conditional execution based on the value of a specified condition. If the condition is true, the `if` body is executed. If not, control flows to the statement following the `else` if there is one or to the statement following the `if` if there is no `else`.

iostream library type providing stream-oriented input and output.

istream Library type providing stream-oriented input.

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

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

manipulator Object, such as `std::endl`, that when read or written “manipulates” the stream itself. Section A.3.1 (p. 825) covers manipulators in more detail.

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

method Synonym for member function.

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

ostream Library type providing stream-oriented output.

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

preprocessor directive An instruction to the C++ preprocessor. `#include` is a preprocessor directive. Preprocessor directives must appear on a single line. We'll learn more about the preprocessor in Section 2.9.2.

return type Type of the value returned by a function.

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

standard error An output stream intended for use for error reporting. Ordinarily, on a windowing operating system, the standard output and the standard error are tied to the window in which the program is executed.

standard input The input stream that ordinarily is associated by the operating system with the window in which the program executes.

standard library Collection of types and functions that every C++ compiler must support. The library provides a rich set of capabilities including the types that support IO. C++ programmers tend to talk about “the library,” meaning the entire standard library or about particular parts of the library by referring to a library type. For example, programmers also refer to the “`iostream` library,” meaning the part of the standard library defined by the `iostream` classes.

standard output The output stream that ordinarily is associated by the operating system with the window in which the program executes.

statement The smallest independent unit in a C++ program. It is analogous to a sentence in a natural language. Statements in C++ generally end in semicolons.

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

string literal Sequence of characters enclosed in double quotes.

uninitialized variable Variable that has no initial value specified. There are no uninitialized variables of class type. Variables of class type for which no initial value is specified are initialized as specified by the class definition. You must give a value to an uninitialized variable before attempting to use the variable's value. *Uninitialized variables can be a rich source of bugs.*

variable A named object.

while statement An iterative control statement that executes the statement that is the `while` body as long as a specified condition is true. The body is executed zero or more times, depending on the truth value of the condition.

() operator The call operator: A pair of parentheses “`()`” following a function name. The operator causes a function to be invoked. Arguments to the function may be passed inside the parentheses.

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

+= operator A compound assignment operator. Adds right-hand operand to the left and stores the result back into the left-hand operand; `a += b` is equivalent to `a = a + b`.

. operator Dot operator. Takes two operands: the left-hand operand is an object and

the right is the name of a member of that object. The operator fetches that member from the named object.

:: operator Scope operator. We'll see more about scope in Chapter 2. Among other uses, the scope operator is used to access names in a namespace. For example, `std::cout` says to use the name `cout` from the namespace `std`.

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

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

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

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

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

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

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

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

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

Index

Bold face numbers refer to the page on which the term was first defined.
Numbers in *italic* refer to the “Defined Terms” section in which the term is defined.

- ... (ellipsis parameter), 244
- /* */ (block comment), **10**, 28
- // (single-line comment), **10**, 28
- _DATE_**, 221
- _FILE_**, 221
- _LINE_**, 221
- _TIME_**, 221
- _cplusplus**, 803
- \0 (null character), 40
- \xnm (hexadecimal escape sequence), 40
- \n (newline character), 40
- \t (tab character), 40
- { } (curly brace), **3**, 28
- #define, 71
- #ifdef, 71
- #ifndef, 71
- #include, 7
- classname**, see destructor
- L'c' (wchar_t literal), 40
- ctrl-d (Unix end-of-file), 20
- ctrl-z (Windows end-of-file), 20
- ; (semicolon), **3**
 - class definition, 440
- ++ (increment), **13**, 30, 146, 190
 - and dereference, 163
 - iterator, 98, 108, 312
 - overloaded operator, 527
 - pointer, 114
 - prefix yields lvalue, 162
 - reverse iterator, 412
- (decrement), 190
 - and dereference, 163
 - iterator, 312
 - overloaded operator, 527
 - prefix yields lvalue, 162
 - reverse iterator, 412
- * (dereference), **98**, 108, 146, 389
 - iterator, 98
- on map yields pair, 362
- overloaded operator, 524
- pointer, 119
 - yields lvalue, 99, 120
- & (address-of), 115, **146**, 511
- > (arrow operator), 164
 - class member access, 445
 - overloaded operator, 525
- >* (pointer to member arrow), 783
- [] (subscript), **87**, 108, 146
 - bitset, 105
 - deque, 325
 - map, 363
 - string, 87
 - vector, 94, 325
 - and multi-dimensioned array, 142
 - and pointer, 124
 - array, 113
 - overloaded operator, 522
 - reference return, 522
 - valid subscript range, 88
 - yields lvalue, 88
- () (call operator), **25**, 30, 226, 280
 - overloaded operator, 530
- :: (scope operator), **8**, 30, 78, 108
 - base class members, 569
 - class member, 85, 445
 - container defined type, 317
 - member function definition, 262
 - to override name lookup, 449
- = (assignment), **13**, 30, 159
 - and conversion, 179
 - and equality, 161
 - class assignment operator, 476
 - container, 328
 - overloaded operator, 483, 520
 - and copy constructor, 484
 - check for self-assignment, 490

- Message, 490
multiple inheritance, 737
reference return, 483, 521
rule of three, 485
use counting, 495, 498
valuelike classes, 501
pointer, 120
string, 86
to signed, 37
to unsigned, 37
yields lvalue, 160
+= (compound assignment), 13, 30, 161
 string, 86
 iterator, 313
 overloaded operator, 511
 Sales_item, 521
+(addition), 150
 string, 86
 iterator, 101, 313
 pointer, 123
 Sales_item, 517
-(subtraction), 150
 iterator, 101, 313
 pointer, 123
*(multiplication), 150
/(division), 150
%(modulus), 151
== (equality), 30, 154
 string, 85
 algorithm, 421
 container, 322
 container adaptor, 350
 iterator, 98, 312
 string, 347
!= (inequality), 30, 154
 container, 322
 container adaptor, 350
 iterator, 98, 312
 string, 347
<(less-than), 30, 153
 overloaded and containers, 520
 used by algorithm, 420
<=(less-than-or-equal), 13, 30, 153
>(greater-than), 30, 153
>=(greater-than-or-equal), 30, 153
>>(input operator), 8, 30
 Sales_item, 516
 istream_iterator, 408
 string, 81, 108
 overloaded operator, 515
 must be nonmember, 514
precedence and associativity, 158
<<(output operator), 7, 30
 bitset, 106
 ostream_iterator, 408
 string, 81, 108
 formatting, 514
 overloaded operator, 513
 must be nonmember, 514
precedence and associativity, 158
Sales_item, 514
>>(right-shift), 155, 190
<<(left-shift), 155, 190
&&(logical AND), 152
 operand order of evaluation, 172
 overloaded operator, 511
||(logical OR), 152
 operand order of evaluation, 172
 overloaded operator, 511
&(bitwise AND), 156, 189
 Query, 610
! (logical NOT), 153
| (bitwise OR), 156, 190
 example, 290
 Query, 610
^ (bitwise XOR), 156, 190
~ (bitwise NOT), 155, 189
 Query, 610
,(comma operator), 168, 189
 example, 289
 operand order of evaluation, 172
 overloaded operator, 511
?: (conditional operator), 165, 189
 operand order of evaluation, 172
+(unary plus), 150
-(unary minus), 150
\nnn (octal escape sequence), 40
ddd.dddL or ddd.ddd1 (long double literal), 39
numEnum or numenum (double literal), 39
numF or numf (float literal), 39
numL or numl (long literal), 39
numU or numu (unsigned literal), 39
class member : constant expression, see bit-field

A

- abnormal termination, stream buffers, 292
abort, 692, 748
absInt, 530

- abstract base class, **596**, 621
 - example, 609
- abstract data type, **78**, **429**, 473
- abstraction, data, **432**, 474
- access control, 65
 - in base and derived classes, 570
 - local class, 796
 - nested class, 787
 - using declarations to adjust, 573
- access label, **65**, **73**, 432, 473
 - private, **65**, 432
 - protected, **562**, 622
 - public, **65**, 432
- Account, 468
- accumulate, 396, 823
- Action, 783
- adaptor, **348**, 353
 - container, 348
 - function, **533**, 535, 553
 - iterator, 399
- addition (+), 150
 - string, 86
 - iterator, 101, 313
 - pointer, 123
 - Sales_item, 517
- address, **35**, 73
- address-of (&), **115**, 146
 - overloaded operator, 511
- adjacent_difference, 824
- adjacent_find, 813
- algorithm, **392**, 424
 - _copy versions, 400, 421
 - _if versions, 421
 - element type constraints, 394
 - independent of container, 393
 - iterator argument constraints, 397, 415
 - iterator category and, 416, 418
 - naming convention, 420–421
 - overloaded versions, 420
 - parameter pattern, 419–420
 - passing comparison function, 403
 - read-only, 396
 - structure, 419
 - that reorders elements, 421
 - that writes elements, 398
 - type independence, 394, 396
 - using function object as argument, 531
 - with two input ranges, 420
- algorithm header, 395
- alias, namespace, **720**, 750
- allocator, **755**, 755–759, 805
 - allocate, 759
 - compared to operator new, 761
 - construct, 755, 758
 - compared to placement new, 762
 - deallocate, 759
 - compared to operator delete, 761
 - destroy, 755, 759
 - compared to calling destructor, 763
- operations, 756
- alternative operator name, 46
- ambiguous
 - conversion, 541–544
 - multiple inheritance, 734
 - function call, **269**, 272, 280
 - multiple base classes, 738
 - overloaded operator, 550
- AndQuery, 609
 - definition, 618
 - eval function, 619
- anonymous union, **795**, 805
- app (file mode), 296
- append, string, 342
- argc, 244
- argument, **25**, **28**, **226**, 227, 280
 - array type, 238
 - C-style string, 242
 - const reference type, 235
 - conversion, 229
 - with class type conversion, 541
 - copied, 230
 - uses copy constructor, 478
 - default, 253
 - iterator, 238, 242
 - multi-dimensioned array, 241
 - passing, 229
 - pointer to const, 231
 - pointer to nonconst, 231
 - reference parameter, 233
 - template, *see* template argument
 - to main, 243
 - to member function, 260
 - nonconst reference parameter, 235
 - type checking, 229
 - ellipsis, 244
 - of array type, 239
 - of reference to array, 240
 - with class type conversion, 541
- argument deduction, template, **637**
- argument list, 226

- argv, 244
- arithmetic
 - iterator, 100, 107, 312, 313
 - pointer, 123, 146
- arithmetic operator
 - and compound assignment, 162
 - function object, 533
 - overloaded operator, 517
- arithmetic type, 34, 73
 - conversion, 180, 188
 - from bool, 182
 - signed to unsigned, 36
 - conversion to bool, 182
- array, 40, 73, 110–114
 - and pointer, 122
 - argument, 238
 - as initializer of vector, 140
 - assignment, 112
 - associative, 388
 - conversion to pointer, 122, 181
 - and template argument, 639
 - copy, 112
 - default initialization, 111
 - uses copy constructor, 478
 - uses default constructor, 460
 - definition, 110
 - elements and destructor, 485
 - function returning, 228
 - initialization, 111
 - multi-dimensioned, 141–144
 - and pointer, 143
 - definition, 142
 - initialization, 142
 - parameter, 241
 - subscript operator, 142
 - of char initialization, 112
 - parameter, 238–244
 - buffer overflow, 242
 - convention, 241–243
 - reference type, 240
 - size calculation, 309
 - and sizeof, 167
 - subscript operator, 113
- arrow operator (->), 164
 - auto_ptr, 704
 - class member access, 445
 - generic handle, 670
 - overloaded operator, 525
- assert preprocessor macro, 221, 223
- assign
 - container, 328
 - string, 340
- assignment
 - vs. initialization, 49
 - memberwise, 483, 503
- assignment (=), 13, 30, 159, 502
 - and conversion, 179
 - and copy constructor, 484
 - check for self-assignment, 490
 - container, 328
 - for derived class, 586
 - Message, 490
 - multiple inheritance, 737
 - overloaded operator, 476, 483, 520
 - reference return, 483, 521
 - pointer, 120
 - rule of three, 485
 - exception for virtual destructors, 588
 - string, 86
 - synthesized, 483, 503
 - to base from derived, 578
 - to signed, 37
 - to unsigned, 37
 - use counting, 495, 498
 - usually not virtual, 588
 - valuelike classes, 501
 - yields lvalue, 160
- associative array, *see* map, 388
- associative container, 356, 388
 - assignment (=), 328
 - begin, 369
 - clear, 359
 - constructors, 360
 - count, 377
 - element type constraints, 309, 323
 - empty, 359
 - equal_range, 379
 - erase, 359
 - find, 377
 - insert, 364
 - key type constraints, 360
 - lower_bound, 377
 - operations, 358
 - overriding the default comparison, 604
 - rbegin, 412
 - rend, 412
 - returning an, 381
 - reverse_iterator, 412
 - size, 359
 - supports relational operators, 359

swap, 329
types defined by, 361
upper_bound, 377
associativity, 149, 170, 188
overloaded operator, 507
at
 deque, 325
 vector, 325
ate (file mode), 296
auto_ptr, 702, 748
 constructor, 703
 copy and assignment, 704
 default constructor, 705
 get member, 705
 operations, 703
 pitfalls, 707
 reset member, 706
 self-assignment, 705
automatic object, 255, 280
 see also local variable
 see also parameter
 and destructor, 485

B

back
 queue, 352
 sequential container, 324
back_inserter, 399, 406, 424
bad, 289
bad_alloc, 175, 219
bad_cast, 219, 774
bad_typeid, 776
badbit, 289
base, 414
base class, 285, 302, 558, 621
 see also virtual function
abstract, 596, 621
 example, 609
access control, 561, 571
assignment operator, usually not virtual, 588
can be a derived class, 566
constructor, 581
 calls virtual function, 589
 not virtual, 588
conversion from derived, 567
 access control, 579
definition, 560
destructor
 calls virtual function, 589
 usually virtual, 587
friendship not inherited, 575
handle class, 599
member operator delete, 764
member hidden by derived, 593
multiple, *see* multiple base class
must be complete type, 566
no conversion to derived, 580
object initialized or assigned from derived, 578
scope, 590
static members, 576
user, 563
virtual, 751
 see virtual base class
Basket, 605
 total function, 606
Bear, 731
 as virtual base, 741
begin, 353
 map, 369
 set, 372
 vector, 97
 container, 317
best match, 269, 280
 see also function matching
bidirectional iterator, 417, 424
 list, 417
 map, 417
 set, 417
binary (file mode), 296
binary function object, 533
binary operator, 148, 188
binary_search, 814
BinaryQuery, 609
 definition, 617
bind1st, 535
bind2nd, 535
binder, 535, 552
binding, dynamic, 559, 621
 requirements for, 566
bit-field, 798, 805
 access to, 798
bitset, 101, 101–106, 107
 any, 104
 count, 104
 flip, 105
 compared to bitwise NOT, 155
 none, 104
 reset, 105
 set, 105

size, 104
 test, 105
 to_ulong, 105
 compared to bitwise operator, 156
 constructor, 101–103
 header, 101
 output operator, 106
 subscript operator, 105
 bitwise AND (&), 156, 189
 example, 610
 bitwise exclusive or (^), 156, 190
 bitwise NOT (~), 155, 189
 example, 610
 bitwise operator, 154–159
 and compound assignment, 162
 compared to bitset, 156
 compound assignment, 157
 example, 290
 operand, 155
 bitwise OR (|), 156, 190
 example, 290, 610
 block, 3, 13, 28, 193, 223
 try, 216, 217, 224, 750
 as target of if, 196
 function, 227
 block scope, 193
 body, function, 3, 29, 226, 281
 book finding program
 using equal_range, 379
 using find, 377
 using upper_bound, 378
 bookstore program, 26
 exception classes, 698
 bool, 35
 and equality operator, 154
 conversion to arithmetic type, 182
 literal, 39
 boolalpha manipulator, 826
 brace, curly, 3, 28
 break statement, 212, 223
 and switch, 201–203
 buffer, 8, 28
 flushing, 290
 buffer overflow, 114
 and C-style string, 132
 array parameter, 242
 built-in type, 3, 28, 34–37
 class member default initialization, 264
 conversion, 179
 initialization of, 51
 Bulk_item
 class definition, 564
 constructor, 581
 constructor using default arguments, 582
 derived from Disc_item, 584
 interface, 558
 member functions, 559
 byte, 35, 73

C

C++
 calling C function from C++, 801
 compiling C and C++, 803
 using C linkage, 802
 .C file, 4
 .cc file, 4
 .cpp file, 4
 .cp file, 4
 C library header, 90
 C with classes, 430
 C-style cast, 186
 C-style string, 112, 130, 130–134, 145
 and char*, 131
 and string literal, 140
 compared to string, 134, 138
 definition, 130
 dynamically allocated, 138
 initialization, 130
 parameter, 242
 pitfalls with generic programs, 671
 c_str, 140
 example, 294
 CachedObj
 add_to_freelist, 771
 operator delete, 770
 operator new, 769
 allocation explained, 769
 definition, 767
 definition of static members, 771
 design, 766
 illustration, 767
 inheriting from, 768
 call operator (), 25, 30, 226, 280
 execution flow, 226
 overloaded operator, 530
 calling C function from C++, 801
 candidate function, 270, 280
 and function templates, 679
 namespaces, 728

- overloaded operator, 549
capacity
 string, 336
 vector, 331
case label, 201, 201–204, 223
 default, 203
cassert header, 221
cast, 183, 188
 checked, *see* dynamic_cast
 old-style, 186
catch clause, 216, 217, 223, 693, 749
 catch(...), 696, 748
 example, 217
 exception specifier, 693
 matching, 693
 ordering of, 694
 parameter, 693
category iterator, 425
ctype, 88–89, 107
 header, 88
cerr, 6, 28
char, 34
 literal, 40
char string literal, *see* string literal
character
 newline (\n), 40
 nonprintable, 40, 75
 null (\0), 40
 printable, 88
 tab (\t), 40
checked cast, *see* dynamic_cast
CheckedPtr, 526
children’s story program, 400
 revisited, 531
cin, 6, 28
 by default tied to cout, 292
cl, 4
class, 20, 28, 63, 73, 473
 static member, 474
 abstract base, 621
 example, 609
 access labels, 65, 432
 as friend, 465
 base, 285, 302, *see* base class, 621
 concrete, 433
 conversion, 552
 multiple conversions lead to ambiguities, 546
 conversion constructor, 461
 function matching, 547
 with standard conversion, 540
data member, 64, 74
 const vs. mutable, 443
 const, initialization, 455
 constraints on type, 438
 definition, 435
 initialization, 454
 mutable, 443
 reference, initialization, 455
 static, 469
data member definition, 65
default access label, 433
default inheritance access label, 574
definition, 64, 430–440
 and header, 264, 437
derived, 285, 302, *see* derived class, 621
destructor definition, 486
direct base, *see* immediate base class, 621
explicit constructor, 462
forward declaration, 438
generic handle, 667, 683
handle, *see* handle class, 599, 622
immediate base, 566, 622
indirect base, 566, 622
local, *see* local class, 806
member, 64, 73, 430
member access, 445
member function, 25, 29, *see* member function
member : constant expression, *see* bit-field
multiple inheritance, *see* multiple base class
nested, *see* nested class, 806
nonvirtual function, calls resolved at compile time, 569
operator delete, *see* member operator
operator new, *see* member operator new
pointer member
 copy control, 492–501
 copy control strategies, 499
 default copy behavior, 493
pointer to member, 780
 definition, 781
pointer to member function, definition, 782
preventing copies, 481
private member, 431

- inheritance, 561
- private member, 75
- protected member, 562
- public member, 75, 431
 - inheritance, 561
- static member, 467
 - as default argument, 471
 - data member as constant expression, 471
 - example, 468
 - inheritance, 576
- template member, *see* member template
- type member, 435
- undefined member, 482
- user, 433, 563
- virtual base, 751
- virtual function, calls resolved at run time, 569
- class, keyword, 64
 - compared to typename, 631
 - in template parameter, 630
 - in variable definition, 440
- class keyword, 473
- class declaration, 438, 473
 - of derived class, 566
- class derivation list, 563, 621
 - access control, 570
 - default access label, 574
 - multiple base classes, 731
 - virtual base, 742
- class scope, 65, 444, 473
 - friend declaration, 466
 - inheritance, 590–595
 - member definition, 445
 - name lookup, 447
 - static members, 470
 - virtual functions, 594
- class template, 90, 107, 627, 683
 - see also* template parameter
 - see also* template argument
 - see also* instantiation
 - compiler error detection, 634
 - declaration, 629
 - definition, 627
 - error detection, 635
 - explicit template argument, 636
 - export, 645
 - friend
 - declaration dependencies, 658
 - explicit template instantiation, 657
 - nontemplate class or function, 656
 - template class or function, 656
- member function, 653
 - defined outside class body, 651
 - instantiation, 653
- member specialization, 677
- member template, *see* member template
- nontype template parameter, 655
- static member, 665
 - accessed through an instantiation, 666
 - definition, 666
- type includes template argument(s), 628, 637
- type-dependent code, 634
- uses of template parameter, 649
- class template specialization
 - definition, 675
 - member, declaration, 677
 - member defined outside class body, 676
- namespaces, 730
- class type, 20, 28, 65
- class member default initialization, 264
- conversion, 183
- initialization of, 52
- object definition, 439
- parameter and overloaded operator, 507
- variable vs. function declaration, 460
- cleanup, object, *see* destructor
- clear, 289, 290
 - associative container, 359
 - example, 290, 295
 - sequential container, 327
- clog, 6, 28
- close, 294
- comma operator (,), 168, 189
 - example, 289
 - operand order of evaluation, 172
 - overloaded operator, 511
- comment, 10, 28
 - block /* */, 10, 28
 - single-line //, 10, 28
- compare
 - plain function, 624
 - string, 347
 - template version, 625
 - instantiated with pointer, 671

- specialization, 672
- compilation
 - and header, 67
 - conditional, 220
 - inclusion model for templates, **644**
 - needed when class changes, 434
 - needed when inline function changes, 258
 - separate, **67**, 76
 - of templates, 643
 - separate model for templates, **644**
- compiler
 - extension, **112**
 - flag for inclusion compilation model, 645
 - GNU, 4
 - Microsoft, 4
 - template errors diagnosed at link time, 635
- compiler extension, 145
- compiling C and C++, 803
- composition vs. inheritance, 573
- compound assignment (e.g., `+=`), **13**, 30, 161
 - string, 86
 - bitwise operator, 157
 - iterator, 313
 - overloaded operator, 511, 518
 - `Sales_item`, 521
- compound expression, **168**, 188
- compound statement, **193**, 223
- compound type, **58**, 73, 145
- compute, 542
 - overloaded version, 545
- concatenation
 - Screen operations, 441
 - string, 86
 - string literal, 41
- concrete class, **433**
 - initialization, 464
- condition, **13**, 28
 - and conversion, 179
 - assignment in, 161
 - in `do while` statement, 211
 - in `for` statement, 15, 207
 - in `if` statement, 18, 195
 - in logical operator, 152
 - in `while` statement, 205
 - stream type as, 19, 183, 288
 - string input operation as, 82
- condition state, **287**, 302
- conditional compilation, 220
- conditional operator (`? :`), **165**, 189
 - operand order of evaluation, 172
- console window, 6
- const, 57
 - and dynamically allocated array, 136
 - conversion to, 182, 231
 - and template argument, 639
 - iterator vs. `const_iterator`, 100
 - object scope, 57, 69
 - overloading and, 267, 275
 - parameter, 231
 - pointer, 128
 - reference, **59**
 - initialization, 60
- const object, constructor, 453
- const data member
 - static data member, 470
 - compared to mutable, 443
 - initialization, 455
- const member function, **261**, **262**, 280, 431, 473
 - overloaded, 442
 - reference return, 442
 - static, 469
- const pointer, *see also* pointer to const
 - conversion from nonconst, 182
- const reference
 - argument, 235
 - conversion from nonconst, 182
 - parameter, 235
 - overloading, 275
 - return type, 249
- const void*, **127**, 145
- const_cast, **183**, 184
- const_iterator, 99, 415
 - compared to const iterator, 100
 - container, 316
- const_reference, 317
- const_reverse_iterator, 412
 - container, 316
- constant expression, **62**, 74
 - and header file, 69
 - array index, 110
 - bit-field, 798
 - enumerator, 62
 - nontype template parameter, 633
 - static data member, 471
- construction, order of, 456, 749
 - derived objects, 581, 582
 - multiple base classes, 732
 - virtual base classes, 746

- constructor, **49**, **74**, **262**, **281**, **431**
 const objects, **453**
 conversion, **461**, **474**
 function matching, **547**
 with standard conversion, **540**
 copy, **476**–**482**, **502**
 base from derived, **578**
 multiple inheritance, **737**
 default, **52**, **74**, **262**, **281**, **458**–**461**, **474**
 default argument in, **458**
 derived class, **581**
 initializes immediate base class, **583**
 initializes virtual base, **744**
 execution flow, **454**
 explicit, **462**, **474**
 copy-initialization, **477**
 for associative container, **360**
 for sequential container, **307**–**309**
 function try block, **696**
 in constructor initializer list, **457**
 inheritance, **581**
 initializer, **452**
 may not be virtual, **588**
 object creation, **452**
 order of construction, **456**
 derived objects, **581**, **582**
 multiple base classes, **732**
 virtual base classes, **746**
 overloaded, **452**
 pair, **357**
 resource allocation, **700**
 synthesized copy, **479**, **503**
 synthesized default, **264**, **281**, **459**, **474**
 virtual inheritance, **744**
- constructor initializer list, **263**, **281**, **431**, **453**–**458**, **474**
 compared to assignment, **454**
 derived classes, **582**
 function try block, **696**
 initializers, **457**
 multiple base classes, **733**
 sometimes required, **455**
 virtual base class, **745**
- container, **90**, **107**, **306**, **353**
see also sequential container
 see also associative container
 and generic algorithms, **393**
 as element type, **311**
 assignment (=), **328**
 associative, **356**, **388**
 begin, **317**
- clear, **327**
 const_iterator, **316**
 const_reference, **317**
 const_reverse_iterator, **316**
 element type constraints, **309**, **323**
 elements and destructor, **485**
 elements are copies, **318**
 empty, **323**
 end, **317**
 erase, **402**
 has bidirectional iterator, **417**
 inheritance, **597**
 insert, **319**
 iterator, **316**
 rbegin, **317**, **412**
 reference, **317**
 rend, **317**, **412**
 returning a, **381**
 reverse_iterator, **316**, **412**
 sequential, **306**, **354**
 size, **323**
 size_type, **316**
 supports relational operators, **321**
 swap, **329**
 types defined by, **316**
- continue statement, **214**, **223**
 example, **290**
- control, flow of, **192**, **224**
- conversion, **178**, **188**
 ambiguous, **541**–**544**
 and assignment, **159**
 argument, **229**
 with class type conversion, **541**
 arithmetic type, **180**, **188**
 array to pointer, **122**, **238**
 and template argument, **639**
- class type, **183**, **535**, **552**
 design considerations, **544**
 example, **537**
 multiple conversions lead to ambiguities, **546**
 operator, **537**, **537**–**540**, **552**
 operator and function matching, **545**
 used implicitly, **538**
 with standard conversion, **539**
- constructor, **461**
 function matching, **547**
 with standard conversion, **540**
- derived to base, **567**, **580**
 access control, **579**

- enumeration type to integer, 182
from `istream`, 183
function matching of template and nontemplate functions, 681
function to pointer, 277
and template argument, 639
implicit, 189
inheritance, 577
integral promotion, 180
multi-dimensioned array to pointer, 143
multiple inheritance, 734
nontemplate type argument, 640
of return value, 246
rank for function matching, 272
rank of class type conversions, 545
signed type, 180
signed to unsigned, 36
template argument, 638
to `const` pointer, 127
to `const`, 182
and template argument, 639
parameter matching, 231
virtual base, 743
conversion constructor, 474
`copy`, 815
`copy` constructor, 476, 476–482, 502
and assignment operator, 484
argument passing, 478
base from derived, 578
for derived class, 586
initialization, 478
`Message`, 489
parameter, 480
pointer members, 480
rule of three, 485
exception for virtual destructors, 588
synthesized, 479, 503
use counting, 495, 497
valuelike classes, 500
`copy` control, 476, 502
handle class, 601
inheritance, 584–590
message handling example, 489
multiple inheritance, 737
of pointer members, 499
`copy-initialization`, 48
using constructor, 477
`copy_backward`, 816
`count`, use, 495, 503
count, 812
book finding program, 377
`map`, 367
`multimap`, 377
`multiset`, 377
`set`, 372
`count_if`, 404, 812
with function object argument, 532
`cout`, 6, 28
by default tied to `cin`, 292
`cstdint` header, 104, 123
`cstdlib` header, 247
`cstring` header, 132
curly brace, 3, 28
- ## D
- dangling `else`, 198, 223
dangling pointer, 176, 188
returning pointer to local variable, 249
synthesized copy control, 494
data abstraction, 432, 474
advantages, 434
data hiding, 434
data structure, 20, 28
data type, abstract, 473
`dec` manipulator, 827
decimal literal, 38
declaration, 52, 74
class, 438, 473
class template member specialization, 677
dependencies and template friends, 658
derived class, 566
`export`, 645
forward, 438, 474
function, 251
exception specification, 708
function template specialization, 672, 673
member template, 661
template, 629
`using`, 78, 108, 720, 750
access control, 573
class member access, 574
overloaded inherited functions, 593
declaration statement, 193, 224
decrement (–), 190
iterator, 312

overloaded operator, 526
prefix yields lvalue, 162
reverse iterator, 412
deduction, template argument, 637
default argument, 253
and header file, 254
function matching, 270
in constructor, 458
initializer, 254
overloaded function, 267
virtual functions, 570
default case label, 203, 224
default constructor, 52, 74, 262, 281, 458–
 461, 474
 Sales_item, 263
 string, 52, 81
 default argument, 458
 synthesized, 264, 281, 459, 474
 used implicitly, 459
 variable definition, 460
definition, 52, 74
 array, 110
 base class, 560
 C-style string, 130
 class, 64, 430–440
 class data member, 65, 435
 class static member, 469
 class template, 627
 static member, 666
 class template specialization, 675
 member defined outside class body,
 676
 class type object, 439
 derived class, 563
 destructor, 486
 dynamically allocated array, 135
 dynamically allocated object, 174
 function, 3
 inside an `if` condition, 196
 inside a `switch` expression, 203
 inside a `while` condition, 205
 map, 360, 373
 multi-dimensioned array, 142
 namespace, 712
 can be discontiguous, 714
 member, 716
 of variable after case label, 204
 overloaded operator, 482
 pair, 356
 pointer, 115
 pointer to function, 276
 static data member, 470
 variable, 48
delete, 145, 176, 188, 806
 compared to `operator delete`, 760
 const object, 178
 execution flow, 760
 member operator, 806
 member operator
 and inheritance, 764
 interface, 764
 memory leak, 177, 485
 null pointer, 176
 runs destructor, 485
delete [], 135
 and dynamically allocated array, 137
deque, 353
 as element type, 311
 assign, 328
 assignment (=), 328
 at, 325
 back, 324
 begin, 317
 clear, 327
 const_iterator, 316
 const_reference, 317
 const_reverse_iterator, 316
 constructor from element count, uses
 copy constructor, 478
 constructors, 307–309
 difference_type, 316
 element type constraints, 309, 323
 empty, 323
 end, 317
 erase, 326
 invalidates iterator, 326
 front, 324
 insert, 319
 invalidates iterator, 320
 iterator, 316
 iterator supports arithmetic, 312
 performance characteristics, 334
 pop_back, 326
 pop_front, 326
 push_back, 318
 invalidates iterator, 321
 push_front, 318
 invalidates iterator, 321
 random-access iterator, 417
 rbegin, 317, 412
 reference, 317
 relational operators, 321

- rend, 317, 412
- resize, 323
- reverse_iterator, 316, 412
- size, 323
- size_type, 316
- subscript ([]), 325
- supports relational operators, 313
- swap, 329
- types defined by, 316
- value_type, 317
- dereference (*), **98**, 108, 146, 389
 - and increment, 163
 - auto_ptr, 704
 - iterator, 98
 - on map iterator yields pair, 362
 - overloaded operator, 524
 - pointer, 119
 - yields lvalue, 99, 120
- derivation list, class, **563**, 621
 - access control, 570
 - default access label, 574
- derived class, **285**, 302, **558**, 621
 - see also* virtual function
 - access control, 561, 572
 - as base class, 566
 - assigned or copied to base object, 578
 - assignment (=), 586
 - constructor, 581
 - calls virtual function, 589
 - for remote virtual base, 744
 - initializes immediate base class, 583
 - constructor initializer list, 582
 - conversion to base, 567
 - access control, 579
 - copy constructor, 586
 - default derivation label, 574
 - definition, 563
 - destructor, 587
 - calls virtual function, 589
 - friendship not inherited, 576
 - handle class, 599
 - member operator delete, 764
 - member hides member in base, 593
 - multiple base classes, 731
 - no conversion from base, 580
 - scope, 590
 - scope (:) to access base class member, 569
 - static members, 576
 - using declaration
 - inherited functions, 593
- member access, 574
- with remote virtual base, 742
- derived object
 - contains base part, 565
 - multiple base classes, contains base part for each, 732
- design
 - CachedObj, 766
 - class member access control, 563
 - class type conversions, 544
 - consistent definitions of equality and relational operators, 520
 - is-a relationship, 573
 - Message class, 486
 - namespace, 714
 - of handle classes, 599
 - of header files, 67
 - export, 646
 - inclusion compilation model, 644
 - separate compilation model, 645
- optimizing new and delete, 764
 - using freelist, 766
- overloaded operator, 510–513
- overview of use counting, 495
- Query classes, 609–611
- Queue, 647
- resource allocation is initialization, 700–701
- Sales_item handle class, 599
- TextQuery class, 380
- vector memory allocation strategy, 756
- writing generic code, 634
- pointer template argument, 671
- destruction, order of, 749
 - derived objects, 587
 - multiple base classes, 733
 - virtual base classes, 747
- destructor, **476**, 484, 502
 - called during exception handling, 691
 - container elements, 485
 - definition, 486
 - derived class, 587
 - explicit call to, 762
 - implicitly called, 484
 - library classes, 709
 - Message, 491
 - multiple inheritance, 737
 - order of destruction, 485
 - derived objects, 587
 - multiple base classes, 733

- virtual base classes, 747
 - resource deallocation, 700
 - rule of three, 485
 - exception for virtual destructors, 588
 - should not throw exception, 692
 - synthesized, 485, 486
 - use counting, 495, 497
 - valuelike classes, 500
 - virtual, multiple inheritance, 736
 - virtual in base class, 587
 - development environment, integrated, 3
 - difference_type**, 101, 107, 316
 - dimension, 110, 145
 - direct base class, *see* immediate base class, 621
 - direct-initialization, 48
 - using constructor, 477
 - directive, using, 721, 751
 - pitfalls, 724
 - Disc_item**, 583
 - class definition, 583
 - discriminant, 794, 806
 - divides<T>**, 534
 - division (/), 150
 - do while statement, 210
 - condition in, 211
 - domain_error, 219
 - dot operator (.), 25, 30
 - class member access, 445
 - double, 37
 - literal (*numEnum or numenum*), 39
 - long double, 37
 - notation output format control, 830
 - output format control, 828
 - duplicate word program, 400–404
 - revisited, 531
 - dynamic binding, 559, 621
 - in C++, 569
 - requirements for, 566
 - dynamic type, 568, 622
 - dynamic_cast**, 183, 773, 806
 - example, 773
 - throws **bad_cast**, 774
 - to pointer, 773
 - to reference, 774
 - dynamically allocated, 145
 - const object, 177
 - array, 134, 134–139
 - delete, 137
 - definition, 135
 - initialization, 136
 - of const, 136
 - C-style string, 138
 - memory and object construction, 754
 - object, 174
 - auto_ptr**, 702
 - constructor, 453
 - destructor, 485
 - exception, 700
- E**
- edit-compile-debug, 16, 28
 - errors at link time, 635
 - else**, *see if* statement
 - dangling, 198, 223
 - empty
 - string, 83, 107
 - vector, 93, 107
 - associative container, 359
 - container, 323
 - priority_queue, 352
 - queue, 352
 - stack, 351
 - encapsulation, 432, 474
 - advantages, 434
 - end, 353
 - map, 369
 - set, 372
 - vector, 97
 - container, 317
 - end-of-file, 19, 28, 835
 - entering from keyboard, 20
 - Endangered, 731
 - endl**, 8
 - manipulator flushes the buffer, 291
 - ends, manipulator flushes the buffer, 291
 - enum keyword, 62
 - enumeration, 62, 74
 - conversion to integer, 182
 - function matching, 274
 - enumerator, 62, 74
 - conversion to integer, 182
 - environment, integrated development, 3
 - eof**, 289
 - eofbit**, 289
 - equal**, 814
 - equal** member function, 778
 - equal_range**, 814
 - associative container, 379
 - book finding program, 379

- equal_to<T>, 534
equality (==), 30, 154
 string, 85
 algorithm, 421
 and assignment, 161
 container, 322
 container adaptor, 350
 iterator, 98, 312
 overloaded operator, 512, 518
 consistent with equality, 520
 string, 347
- erase
 associative container, 359
 container, 402
 invalidates iterator, 326
 map, 368
 multimap, 376
 multiset, 376
 sequential container, 326
 set, 372
 string, 340
- error, standard, 6
escape sequence, 40, 74
 hexadecimal (\xnnn), 40
 octal (\nnn), 40
- evaluation
 order of, 149, 189
 short-circuit, 152
- exception, raise, *see throw*
- exception
 class, 216, 224
 class hierarchy, 698
 constructor, 220
 extending the hierarchy, 697
 header, 219
 what member, 218, 697
- exception handling, 215–220, 749
 see also throw
 see also catch clause
 and terminate, 219
 compared to assert, 221
 exception in destructor, 692
 finding a catch clause, 693
 function try block, 696, 749
 handler, *see catch clause*
 library class destructors, 709
 local objects destroyed, 691
 specifier, 217, 224, 693, 749
 nonreference, 693
 reference, 694
 types related by inheritance, 694
- stack unwinding, 691
uncaught exception, 692
unhandled exception, 219
- exception object, 690, 749
 array or function, 689
 initializes catch parameter, 693
 must be copyable, 689
 pointer to local object, 690
 rethrow, 695
- exception safety, 700, 749
- exception specification, 706, 749
 throw(), 708
 function pointers, 711
 unexpected, 708
 violation, 708
 virtual functions, 710
- executable file, 4
- EXIT_FAILURE, 247
EXIT_SUCCESS, 247
- explicit constructor, 462, 474
 copy-initialization, 477
- export, 645
 and header design, 646
 keyword, 645, 683
- exporting C++ to C, 802
- expression, 7, 28, 148, 189
 and operand conversion, 179
 compound, 168, 188
 constant, 62, 74
 throw, 689, 750
- expression statement, 192, 224
- extended_compute, 542
- extension, compiler, 145
- extern, 53
extern 'C', *see linkage directive*
- extern const, 57
- F**
- factorial program, 250
- fail, 289
- failbit, 289
- file
 executable, 4
 object, 68
 source, 4, 29
- file mode, 296, 302
 combinations, 298
 example, 299
- file static, 719, 749
- fill, 816

fill_n, 815
 find, 392, 812

- book finding program, 377
- map, 368
- multimap, 377
- multiset, 377
- set, 372
- string, 344

 find last word program, 414
 find_first_of, 812
 find_first_not_of, string, 346
 find_end, 812
 find_first_of, 396, 812

- string, 345

 find_if, 421, 812
 find_last_not_of, string, 346
 find_last_of, string, 346
 find_val program, 234
 fixed manipulator, 830
 float, 37

- literal (*numF* or *numf*), 39

 floating point

- notation output format control, 830
- output format control, 828

 floating point literal, *see* double literal
 flow of control, 192, 224
 flush, manipulator flushes the buffer, 291
 Folder, *see* Message
 for statement, 29, 207

- condition in, 207
- execution flow, 208
- expression, 207
- for header, 207
- initialization statement, 207
- scope, 15

 for statement for statement, 14
 for_each, 813
 format state, 825
 forward declaration of class type, 438
 forward iterator, 417, 424
 fp_compute, 542
 free store, 135, 145
 freelist, 766, 806
 friend, 465, 474

- class, 465
- class template
 - explicit template instantiation, 657
 - nontemplate class or function, 656
 - template class or function, 656
- function template, example, 659
- inheritance, 575

 member function, 466
 overloaded function, 467
 overloaded operator, 509
 scope considerations, 466

- namespaces, 727

 template example, 658
 friend keyword, 465
 front

- queue, 352
- sequential container, 324

 front_inserter, 406, 424

- compared to inserter, 406

 fstream, 285, 293–299, 302

- see also* istream
- see also* ostream
- close, 294
- constructor, 293
- file marker, 838
- file mode, 296
 - combinations, 298
 - example, 299
- file random access, 838
- header, 285, 293
- off_type, 839
- open, 293
- pos_type, 839
- random IO sample program, 840
- seek and tell members, 838–842

 function, 2, 29, 225, 281

- equal member, 778
- inline, 257, 281
- candidate, 270, 280
- conversion to pointer, 277
 - and template argument, 639
- function returning, 228
- inline and header, 257
- member, 25, 29, *see* member function, 474
- nonvirtual, calls resolved at compile time, 569
- overloaded, 265, 281
 - compared to redeclaration, 266
 - friend declaration, 467
 - scope, 268
 - virtual, 593
- pure virtual, 596, 622
 - example, 609
- recursive, 249, 281
- viable, 270, 282
- virtual, 559, 566–570, 622
 - assignment operator, 588

- calls resolved at run time, 568
- compared to run-time type identification, 777
- default argument, 570
- derived classes, 564
- destructor, 587
- destructor and multiple inheritance, 736
- exception specifications, 710
- in constructors, 589
- in destructor, 589
- introduction, 561
- multiple inheritance, 735
- no virtual constructor, 588
- overloaded, 593
- overloaded operator, 615
- overriding run-time binding, 570
- return type, 564
- run-time type identification, 772
- scope, 594
- to copy unknown type, 602
- type-sensitive equality, 778
- function adaptor, 533, 535, 553
 - bind1st, 535
 - bind2nd, 535
 - binder, 535
 - negator, 535
 - not1, 535
 - not2, 535
- function body, 3, 29, 226, 281
- function call
 - ambiguous, 269, 272
 - execution flow, 226
 - overhead, 257
 - through pointer to function, 278
 - through pointer to member, 784
 - to overloaded operator, 509
 - to overloaded postfix operator, 529
 - using default argument, 253
- function declaration, 251
 - and header file, 252
 - exception specification, 708
- function definition, 3
- function matching, 269, 281
 - and overloaded function templates, 679–682
 - examples, 680
- argument conversion, 269
- conversion operator, 545
- conversion rank, 272
 - class type conversions, 545
- enumeration parameter, 274
- integral promotion, 273
- multiple parameters, 272
- namespaces, 727
- of member functions, 436
- overloaded operator, 547–551
- function name, 3, 29
- function object, 531, 553
 - algorithms, 531
 - example, 534
- binary, 533
- library defined, 533
- unary, 533
- function pointer, 276–279
 - and template argument deduction, 640
 - definition, 276
 - exception specifications, 711
 - function returning, 228
 - initialization, 277
 - overloaded functions, 279
 - parameter, 278
 - return type, 278
 - typedef, 276
- function prototype, 251, 281
- function return type, 226, 227, 281
 - const reference, 249
 - no implicit return type, 228
 - nonreference, 247
 - uses copy constructor, 478
 - reference, 248
 - reference yields lvalue, 249
 - void, 245
- function scope, 227
- function table, 785
 - pointer to member, 785
- function template, 625, 683
 - see also* template parameter
 - see also* template argument
 - see also* instantiation
 - as friend, 659
 - compiler error detection, 634
 - declaration, 629
 - error detection, 635
 - explicit template argument, 642
 - and function pointer, 643
 - specifying, 642
 - export, 645
 - inline, 626
 - instantiation, 637
 - template argument deduction, 637

type-dependent code, 634
 function template specialization
 compared to overloaded function, 673
 declaration, 672, 673
 example, 672
 namespaces, 730
 scope, 674
 function try block, 696, 749

G

g++, 4
 gcd program, 226
 generate, 816
 generate_n, 815
 generic algorithm, *see* algorithm
 generic handle class, 667, 683
 generic memory management, *see* CachedObj
 generic programming, 95, 624
 and pointer template argument, 671
 type-independent code, 634
 getline, 82, 107
 example, 300, 386
 global namespace, 716, 750
 global scope, 54, 74
 global variable, lifetime, 254
 GNU compiler, 4
 good, 289
 goto statement, 214, 224
 greater-than (>), 30, 153
 greater-than-or-equal (>=), 30, 153
 greater<T>, 534
 greater_equal<T>, 534
 GT6 program, 403
 GT_cls, 532
 guard header, 71, 74

H

.h file, 21
 Handle, 667
 int instantiation, 668
 operations, 668
 Sales_item instantiation, 669
 handle class, 599, 622
 copy control, 601
 copying unknown type, 602
 design, 599
 generic, 667, 683
 that hides inheritance hierarchy, 610
 using a, 603

handler, *see* catch clause
 has-a relationship, 573
 HasPtr
 as a smart pointer, 495
 using synthesized copy control, 493
 with value semantics, 499
 header, 7, 29, 67, 74
 bitset, 101
 cctype, 88, 107
 cstddef, 104
 iomanip, 829
 string, 80
 vector, 90
 algorithm, 395
 and constant expression, 69
 and library names, 810
 C library, 90
 cassert, 221
 class definition, 264, 437
 cstddef, 123
 cstdlib, 247
 cstring, 132
 default argument, 254
 deque, 307
 design, 67
 export, 646
 inclusion compilation model, 644
 namespace members, 714
 separate compilation model, 645
 exception, 219
 fstream, 285, 293
 function declaration, 252
 inline function, 257
 inline member function definition, 437
 iostream, 285
 iterator, 399
 list, 307
 map, 360, 375
 new, 219
 numeric, 395
 programmer-defined, 67–72
 queue, 349
 Sales_item, 21, 67, 264
 set, 373, 375
 sstream, 285, 300
 stack, 349
 stdexcept, 217, 219
 type_info, 219
 using declaration, 80
 utility, 356

vector, 307
header file, naming convention, 264
header guard, 71, 74
heap, 135, 145
hex manipulator, 827
hexadecimal, literal (`0Xnum` or `0xnum`), 38
hexadecimal escape sequence (`\Xnm`), 40
hides, names in base hidden by names in derived, 592
hierarchy, inheritance, 558, 566, 622
high-order bits, 102, 107

I

IDE, 3
identification, run-time type, 772–780, 807
identifier, 46, 74
 naming convention, 47
 reserved, 47
if statement, else branch, 224
if statement, 17, 29, 195, 224
 compared to switch, 199
 dangling else, 198
 else branch, 18, 197
ifstream, 285, 293–299
 see also istream
 close, 294
 constructor, 293
 file marker, 838
 file mode, 296
 combinations, 298
 example, 299
 file random access, 838
 off_type, 839
 open, 293
 pos_type, 839
 random IO sample program, 840
 seek and tell members, 838–842
immediate base class, 566, 622
implementation, 63, 74, 432
implementation inheritance, 573
implicit this pointer, overloaded operator, 508
implicit conversion, *see* conversion, 189
implicit return, 245
 from main allowed, 247
implicit this pointer, 260, 281, 431, 440
 in and overloaded operator, 483
 static member functions, 469
in (file mode), 296

include, *see #include*
includes, 822
inclusion compilation model, 644, 683
incomplete type, 438, 474
 restriction on use, 438, 566, 693
increment (++), 13, 30, 146, 190
 reverse iterator, 412
 and dereference, 163
 iterator, 98, 108, 312
 overloaded operator, 526
 pointer, 114
 prefix yields lvalue, 162
indentation, 14, 197
index, 87, 107
indirect base class, 566, 622
inequality (!=), 30, 154
 iterator, 98
 container, 322
 container adaptor, 350
 iterator, 312
 overloaded operator, 512, 519
 string, 347
inheritance, 284, 302
 containers, 597
 conversions, 577
 default access label, 574
 friends, 575
 handle class, 599
 implementation, 573
 interface, 573
 iostream diagram, 285
multiple, *see* multiple base class, 731
private, 622
 static members, 576
virtual, 741, 751
inheritance hierarchy, 558, 566, 622
inheritance vs. composition, 573
initialization, 9
 vs. assignment, 49
array, 111
array of char, 112
built-in type, 51
C-style string, 130
class data member, 454
class member of built-in type, 264
class member of class type, 264
class type, 52, 452
const static data member, 470
dynamically allocated array, 136
dynamically allocated object, 174
map, 373

memberwise, 479, 503
 multi-dimensioned array, 142
 objects of concrete class type, 464
 pair, 356
 parameter, 229
 pointer, 117–119
 pointer to function, 277
 return value, 247
 value, 92, 108
 variable, 48, 50, 76
 initialization vs. assignment, 456
 initialized, 48, 75
 initializer list, constructor, 263, 281, 431, 453–458, 474
 inline function, 257, 281
 and header, 257
 function template, 626
 member function, 437
 and header, 437
 inner_product, 823
 inplace_merge, 816
 input, standard, 6
 input (>>), 8, 30
 Sales_item, 516
 istream_iterator, 408
 string, 81, 108
 overloaded operator, 515
 error handling, 516–517
 must be nonmember, 514
 precedence and associativity, 158
 input iterator, 416, 424
 insert
 inserter, 406
 invalidates iterator, 320
 map, 364
 multimap, 376
 multiset, 376
 return type from set::insert, 373
 sequential container, 319
 set, 373
 string, 340
 insert iterator, 399, 405, 425
 inserter, 406
 inserter, 425
 compared to front_inserter, 406
 instantiation, 625, 684
 class template, 628, 636, 654
 member function, 653
 nontype parameter, 655
 type, 637
 error detection, 635
 function template, 637
 from function pointer, 640
 nontemplate argument conversion, 640
 nontype template parameter, 633
 template argument conversion, 638
 member template, 663
 nested class template, 788, 791
 on use, 636
 static class member, 665
 int, 34
 literal, 38
 Integral, 539
 integral promotion, 180, 189
 function matching, 273
 integral type, 34, 75
 integrated development environment, 3
 interface, 63, 75, 432
 interface inheritance, 573
 internal manipulator, 832
 interval, left-inclusive, 314, 354
 invalid_argument, 219
 invalidated iterator, 315, 353
 IO stream, *see* stream
 iomanip header, 829
 iostate, 289
 iostream, 6, 29, 285
 see also istream
 see also ostream
 header, 285
 inheritance hierarchy, 740
 seek and tell members, 838
 is-a relationship, 573
 isalnum, 88
 isalpha, 88
 ISBN, 2
 isbn_mismatch, 699
 destructor explained, 709
 iscntrl, 88
 isdigit, 88
 isgraph, 88
 islower, 88
 isprint, 88
 ispunct, 88
 isShorter program, 235, 403
 isspace, 88
 istream, 6, 29, 285
 see also manipulator
 condition state, 287
 flushing input buffer, 290
 format state, 825

gcount, 837
get, 834

- multi-byte version, 836
- returns int, 835, 836

getline, 82, 836
getline, example, 300
ignore, 837
inheritance hierarchy, 740
input (>>), 8

- precedence and associativity, 158

no containers of, 310
no copy or assign, 287
peek, 834
put, 834
putback, 835
read, 837
seek and tell members, 838
unformatted operation, 834

- multi-byte, 837
- single-byte, 834

unget, 835
write, 837
istream_iterator, 407, 425

- and class type, 410
- constructors, 408
- input iterator, 417
- input operator (>>), 408
- limitations, 411
- operations, 409
- used with algorithms, 411

istringstream, 285, 299–301

- see also istream*
- word per line processing, 300, 370
- str, 301
- word per line processing, 386

isupper, 88
isxdigit, 88
Item_base

- class definition, 560
- constructor, 580
- interface, 558
- member functions, 559

iter_swap, 816
iterator, 95, 95–101, 107, 311–316, 354

- argument, 238
- arrow (->), 164
- bidirectional, 417, 424
- compared to reverse iterator, 413, 414
- destination, 399, 419
- equality, 98, 312
- forward, 417, 424

generic algorithms, 394
inequality, 98, 312
input, 416, 424
insert, 399, 405, 425
invalidated, 315, 353
invalidated by

- assign, 328
- erase, 326
- insert, 321
- push_back, 321
- push_front, 321
- resize, 324

off-the-end, 97, 394, 425
operations, 311
output, 416, 425
parameter, 238, 242
random-access, 417, 425
relational operators, 313
reverse, 405, 412–415, 425
stream, 425
iterator, 362, 374

- container, 316

iterator header, 399
iterator arithmetic, 100, 107, 312, 313

- relational operators, 313

iterator category, 416–418, 425

- algorithm and, 416, 418
- bidirectional iterator, 417
- forward iterator, 417
- hierarchy, 417
- input iterator, 416
- output iterator, 416
- random-access iterator, 417

iterator range, 314, 314–316, 354

- algorithms constraints on, 397, 415
- erase, 327
- generic algorithms, 394
- insert, 320

K

key_type, 388

- associative containers, 362

keyword

- enum, 62
- friend, 465
- namespace, 712
- protected, 562
- template, 625
- try, 217
- union, 793

- virtual, 559
 export, 645
 keyword table, 46
 Koenig lookup, 726
- L**
- label
 case, 201, 201–204, 223
 access, 65, 73, 432, 473
 statement, 214
 labeled statement, 214, 224
 left manipulator, 832
 left-inclusive interval, 314, 354
 left-shift (`<<`), 155, 190
 length_error, 219
 less-than (`<`), 30, 153
 overloaded and containers, 520
 used by algorithm, 420
 less-than-or-equal (`<=`), 13, 30, 153
 less<T>, 534
 less_equal<T>, 534
 lexicographical_compare, 823
 library, standard, 5, 29
 library names to header table, 810
 library type, 29
 lifetime, object, 254, 281
 link time errors from template, 635
 linkage directive, 801, 806
 C++ to C, 802
 compound, 802
 overloaded function, 803
 parameter or return type, 804
 pointer to function, 803
 single, 801
 linking, 68, 75
 list, 354
 as element type, 311
 assign, 328
 assignment (`=`), 328
 back, 324
 begin, 317
 bidirectional iterator, 417
 clear, 327
 const_iterator, 316
 const_reference, 317
 const_reverse_iterator, 316
 constructor from element count, uses
 copy constructor, 478
 constructors, 307–309
 element type constraints, 309, 323
 empty, 323
 end, 317
 erase, 326
 front, 324
 insert, 319
 iterator, 316
 merge, 423
 performance characteristics, 334
 pop_back, 326
 pop_front, 326
 push_back, 318
 push_front, 318
 rbegin, 317, 412
 reference, 317
 relational operators, 321
 remove, 423
 remove_if, 423
 rend, 317, 412
 resize, 323
 reverse, 423
 reverse_iterator, 316, 412
 size, 323
 size_type, 316
 specific algorithms, 421
 splice, 423
 swap, 329
 types defined by, 316
 unique, 423
 value_type, 317
 literal, 37, 37–42, 75
 bool, 39
 char, 40
 decimal, 38
 double (`numEnum` or `numenum`), 39
 float (`numF` or `numf`), 39
 hexadecimal (`0Xnum` or `0xnum`), 38
 int, 38
 long (`numL` or `numl`), 38
 long double (`ddd.dddL` or `ddd.dddl`), 39
 multi-line, 42
 octal (`0num`), 38
 string, 9, 30, 40
 unsigned (`numU` or `numu`), 39
 wchar_t, 40
 local class, 796, 806
 access control, 796
 name lookup, 797
 nested class in, 797
 restrictions on, 796
 local scope, 54, 75

- local static object, 255, 281
local variable, 227, 281
 destructor, 485
 lifetime, 254
 reference return type, 248
logic_error, 219
logical AND (&&), 152
 operand order of evaluation, 172
 overloaded operator, 511
logical NOT (!), 153
logical operator, 152
 function object, 533
logical OR (||), 152
 operand order of evaluation, 172
 overloaded operator, 511
logical_and<T>, 534
logical_not<T>, 534
logical_or<T>, 534
long, 34
 literal (*numL* or *num1*), 38
long double, 37
long double, literal (*ddd.dddL* or *ddd.ddd1*), 39
lookup, name, 447, 474
 and templates, 647
 before type checking, 269, 593
 multiple inheritance, 738
 class member declaration, 447
 class member definition, 448, 450
 class member definition, examples, 449
 collisions under inheritance, 591
 depends on static type, 590
 multiple inheritance, 735
 inheritance, 590, 595
 local class, 797
 multiple inheritance, 737
 ambiguous names, 738
 namespace names, 724
 argument-dependent lookup, 726
 nested class, 791
 overloaded virtual functions, 593
 virtual inheritance, 743
low-order bits, 102, 107
lower_bound, 814
 associative container, 377
 book finding program, 378
lvalue, 45, 75
 assignment, 160
 dereference, 99
 function reference return type, 249
prefix decrement, 162
prefix increment, 162
subscript, 88
- ## M
- machine-dependent
 bitfield layout, 798
 char representation, 36
 division and modulus result, 151
 end-of-file character, 20
 iostate type, 288
 linkage directive language, 802
 nonzero return from main, 247
 pre-compiled headers, 67
 random file access, 837
 reinterpret_cast, 185
 representation of enum type, 274
 return from exception what operation, 220
 signed and out-of-range value, 37
 signed types and bitwise operators, 155
 size of arithmetic types, 34
 template compilation optimization, 645
 terminate function, 219
 type_info members, 779
 vector memory allocation size, 331
 volatile implementation, 799
magic number, 56, 75
main, 2, 29
 arguments to, 243
 not recursive, 251
 return type, 3
 return value, 2–5, 247
 returns 0 by default, 247
make_pair, 358
make_plural program, 248
manip, 542
manipulator, 8, 29, 825
 boolalpha, 826
 dec, 827
 fixed, 830
 hex, 827
 internal, 832
 left, 832
 noboolalpha, 827
 noshowbase, 828
 noshowpoint, 832
 noskipws, 833

nouppercase, 828
oct, 827
right, 832
scientific, 830
setfill, 832
setprecision, 829
setw, 832
showbase, 827
showpoint, 831
skipws, 833
uppercase, 828
boolalpha, 826
change format state, 825
dec, 827
endl flushes the buffer, 291
ends flushes the buffer, 291
fixed, 830
flush flushes the buffer, 291
hex, 827
internal, 832
left, 832
noboolalpha, 827
noshowbase, 828
noshowpoint, 832
noskipws, 833
nouppercase, 828
oct, 827
right, 832
scientific, 830
setfill, 832
setprecision, 829
setw, 832
showbase, 827
showpoint, 831
skipws, 833
unitbuf flushes the buffer, 291
uppercase, 828
map, 356, 388
as element type, 311
assignment (=), 328
begin, 369
bidirectional iterator, 417
clear, 359
constructors, 360
count, 367
definition, 360
dereference yields pair, 362
element type constraints, 309
empty, 359
end, 369
equal_range, 379
erase, 359, 368
find, 368
header, 360
insert, 364
iterator, 362
key type constraints, 360
key_type, 362
lower_bound, 377
mapped_type, 362, 388
operations, 358
overriding the default comparison, 604
rbegin, 412
rend, 412
return type from insert, 365
reverse_iterator, 412
size, 359
subscript operator, 363
supports relational operators, 359
swap, 329
upper_bound, 377
value_type, 361
mapped_type, map, multimap, 362
match, best, 269, 280
max, 822
max_element, 822
member, *see also* class member
 mutable data, 474
 pointer to, 780, 807
member function, 25, 29, 431, 474
 const, 280
 equal, 778
 as friend, 466
 base member hidden by derived, 593
 class template, 653
 defined outside class body, 651
 instantiation, 653
 const, 261, 262
 defined outside class body, 261, 431
 definition, 258–262
 in class scope, 445
 name lookup, 448
 name lookup, examples, 449
function template, *see* member template
implicitly inline, 259
inline, 437
 and header, 437
overloaded, 436
overloaded on const, 442
overloaded operator, 483, 508

- pointer to, definition, 782
returning `*this`, 442
`static`, 467
 `this` pointer, 469
undefined, 482
- member operator `delete`, 764, 806
 and inheritance, 764
 example, 769
 `CachedObj`, 770
 interface, 764
- member operator `delete []`, 765
- member operator `new`, 764, 806
 example, 769
 `CachedObj`, 769
 interface, 764
- member operator `new []`, 765
- member template, 660, 684
 declaration, 661
 defined outside class body, 662
 examples, 660
 instantiation, 663
 template parameters, 663
- memberwise assignment, 483, 503
- memberwise initialization, 479, 503
- memory exhaustion, 175
- memory leak, 177, 485
 after exception, 700
- memory management, generic, *see* `CachedObj`
- merge, 816
 list, 423
- Message, 486–491
 assignment operator, 490
 class definition, 488
 copy constructor, 489
 design, 486
 destructor, 491
 `put_Msg_in_Folder`, 489
 `remove_Msg_from_Folder`, 491
- method, *see also* member function, 29
- Microsoft compiler, 4
- min, 822
- min_element, 822
- minus<T>, 534
- mismatch, 814
- mode, file, 296, 302
- modulus (%), 151
- modulus<T>, 534
- multi-dimensioned array, 141–144
 and pointer, 143
 conversion to pointer, 143
 definition, 142
- initialization, 142
- parameter, 241
- subscript operator, 142
- multi-line literal, 42
- multimap, 375, 388
 assignment (=), 328
 begin, 369
 clear, 359
 constructors, 360
 count, 377
 dereference yields pair, 362
 element type constraints, 309
 empty, 359
 equal_range, 379
 erase, 359, 376
 find, 377
 has no subscript operator, 376
 insert, 376
 iterator, 362, 376
 key type constraints, 360
 key_type, 362
 lower_bound, 377
 mapped_type, 362
 operations, 358, 376
 overriding the default comparison, 604
 rbegin, 412
 rend, 412
 return type from insert, 365
 reverse_iterator, 412
 size, 359
 supports relational operators, 359
 swap, 329
 upper_bound, 377
 value_type, 361
- multiple base class, 750
 see also virtual base class
- ambiguities, 738
- ambiguous conversion, 734
- avoiding potential name ambiguities, 738
- conversions, 734
- definition, 731
- destructor usually virtual, 736
- name lookup, 737
- object composition, 732
- order of construction, 732
- scope, 737
 virtual functions, 735
- multiple inheritance, *see* multiple base class, 731

- multiplication (*), 150
 - `multiplies<T>`, 534
 - `multiset`, 375, 388
 - assignment (=), 328
 - `begin`, 372
 - `clear`, 359
 - constructors, 372
 - `count`, 377
 - element type constraints, 309
 - `end`, 372
 - `equal_range`, 379
 - `erase`, 359, 376
 - `find`, 377
 - `insert`, 376
 - iterator, 376
 - key type constraints, 360
 - `lower_bound`, 377
 - `Sales_item`, 605
 - operations, 358, 376
 - overriding the default comparison, 604
 - `rbegin`, 412
 - `rend`, 412
 - return type from `insert`, 373
 - `reverse_iterator`, 412
 - supports relational operators, 359
 - `swap`, 329
 - `upper_bound`, 377
 - example, 607
 - `value_type`, 372
 - mutable data member, 443, 474
- N**
- \n (newline character), 40
 - name lookup, 447, 474
 - and templates, 647
 - before type checking, 269, 593
 - multiple inheritance, 738
 - class member declaration, 447
 - class member definition, 448, 450
 - class member definition, examples, 449
 - collisions under inheritance, 591
 - depends on static type, 590
 - multiple inheritance, 735
 - inheritance, 590, 595
 - local class, 797
 - multiple inheritance, 737
 - ambiguous names, 738
 - namespace names, 724
 - argument-dependent lookup, 726
 - nested class, 791
 - overloaded virtual functions, 593
 - virtual inheritance, 743
 - name resolution, *see* name lookup
 - namespace, 8, 29, 712, 750
 - class friend declaration scope, 727
 - `cplusplus_primer`, 714
 - definition, 712
 - design, 714
 - discontiguous definition, 714
 - function matching, 727
 - global, 716
 - member, 713
 - member definition, 716
 - outside namespace, 716
 - restrictions, 716
 - nested, 717
 - scope, 713–717
 - unnamed, 718
 - local to file, 718
 - replace file static, 719
 - namespace keyword, 712
 - namespace alias, 720, 750
 - namespace pollution, 712, 750
 - naming convention
 - header file, 264
 - source file, 264
 - NDEBUG, 220
 - `negate<T>`, 534
 - negator, 535, 553
 - nested class, 786, 806
 - access control, 787
 - class defined outside enclosing class, 789
 - in class template, 788
 - in local class, 797
 - member defined outside class body, 788
 - name lookup, 791
 - `QueueItem` example, 787
 - relationship to enclosing class, 787, 790
 - scope, 786
 - static members, 790
 - union, 794
 - nested namespace, 717
 - nested type, *see* nested class 786
 - `new`, 145, 174, 189, 806
 - compared to operator `new`, 760
 - execution flow, 760

- header, 219
member operator, 806
member operator, interface, 764
placement, 761, 807
 compared to construct, 762
new [], 135
new failure, 175
next_permutation, 821
noboolalpha manipulator, 827
NoDefault, 459
nonconst reference, 60
 parameter, 232
 limitations, 235
nonportable, 42
nonprintable character, 40, 75
nonreference
 parameter, 230
 uses copy constructor, 478
return type, 247
 uses copy constructor, 478
nontype template parameter, 625, 628, 632, 684
 see also template parameter
 class template, 655
 must be constant expression, 633
nonvirtual function, calls resolved at compile time, 569
noshowbase manipulator, 828
noshowpoint manipulator, 832
noskipws manipulator, 833
not equal, *see* inequality
not1, 535
not2, 535
not_equal_to<T>, 534
NotQuery, 609
 definition, 616
 eval function, 620
nouppercase manipulator, 828
nth_element, 818
NULL, 118
null pointer, 118
 delete of, 176
null statement, 192, 224
null-terminated array, *see* C-style string
number, magic, 56, 75
numeric header, 395
numeric literal
 float (numF or numf), 39
 long (numL or numl), 39
 long double (ddd.dddL or ddd.ddd1), 39
 unsigned (numU or numu), 39
- ## O
- object, 46, 75
 automatic, 255, 280
 function, 553
 is not polymorphic, 569
 local static, 255, 281
 temporary, 247
object cleanup, *see* destructor
object creation
 constructor, 452
order of construction, 456
 derived objects, 581, 582
 multiple base classes, 732
 virtual base classes, 745
order of destruction, 485
 derived objects, 587
 multiple base classes, 733
 virtual base classes, 747
object file, 68
object lifetime, 254, 281
 and destructor, 485
 compared to scope, 254
object-oriented programming, 285, 302, 622
 key ideas in, 558–560
oct manipulator, 827
octal, literal (0num), 38
octal escape sequence (\nnn), 40
off-the-end iterator, 97, 394, 425
 istream_iterator, 408
off-the-end pointer, 125
ofstream, 285, 293–299
 see also ostream
 close, 294
 constructor, 293
 file marker, 838
 file mode, 296
 combinations, 298
 example, 299
 file random access, 838
 off_type, 839
 open, 293
 pos_type, 839
 random IO sample program, 840
 seek and tell members, 838–842
open, 293
open_file, example of, 370, 383
open_file program, 299
operand, 148, 189

- order of evaluation
 - comma operator, 172
 - conditional operator, 172
 - logical operator, 172
- operator, **148**, 189
 - `sizeof`, 167
 - `typeid`, **775**, 807
 - addition (+), 150
 - `string`, 86
 - iterator, 101, 313
 - pointer, 123
 - address-of (&), **115**
 - arrow (->), 164
 - class member access, 445
 - assignment (=), **13**, 30, 159
 - and conversion, 179
 - and equality, 161
 - container, 328
 - multiple inheritance, 737
 - pointer, 120
 - `string`, 86
 - to `signed`, 37
 - to `unsigned`, 37
 - yields lvalue, 160
 - binary, **148**, 188
 - bitwise AND (&), 156
 - bitwise exclusive or (^), 156
 - bitwise not (~), 155
 - bitwise OR (|), 156
 - bitwise OR (|), example, 290
 - call (()), **30**, 226
 - comma (,), **168**
 - operand order of evaluation, 172
 - comma (,), example, 289
 - compound assignment (e.g., +=), iterator, 313
 - compound assignment (e.g., +=), **13**, 30, 161
 - `string`, 86
 - arithmetic, 162
 - bitwise, 162
 - conditional (? :), **165**
 - operand order of evaluation, 172
 - decrement (--)
 - iterator, 312
 - prefix yields lvalue, 162
 - reverse iterator, 412
 - dereference (*), **98**
 - and increment, 163
 - iterator, 98
 - on map yields pair, 362
 - pointer, 119
 - yields lvalue, 99, 120
 - division (/), 150
 - dot (.), **25**, 30
 - class member access, 445
 - equality (==), 30, 154
 - `string`, 85
 - algorithm, 421
 - container, 322
 - container adaptor, 350
 - iterator, 98, 312
 - `string`, 347
 - greater-than (>), 30, 153
 - greater-than-or-equal (>=), 30, 153
 - increment (++), **13**, 30
 - and dereference, 163
 - iterator, 98, 312
 - pointer, 114
 - prefix yields lvalue, 162
 - reverse iterator, 412
 - inequality (!=), 30, 154
 - container, 322
 - container adaptor, 350
 - iterator, 98, 312
 - `string`, 347
 - input (>>), **8**, 30
 - `Sales_item`, 516
 - `istream_iterator`, 408
 - `string`, 81
 - precedence and associativity, 158
 - left-shift (<<), 155, 190
 - less-than (<), 30, 153
 - used by algorithm, 420
 - less-than-or-equal (<=), **13**, 30, 153
 - logical AND (&&), 152
 - operand order of evaluation, 172
 - logical NOT (!), 153
 - logical OR (||), 152
 - operand order of evaluation, 172
 - modulus (%), 151
 - multiplication (*), 150
 - output (<<), **7**, 30
 - `bitset`, 106
 - `ostream_iterator`, 408
 - `string`, 81
 - precedence and associativity, 158
 - overloaded, 189, **482**, 503
 - pointer to member
 - arrow (->*), 783
 - dot (.*), 783
 - right-shift (>>), 155, 190

- scope (::), 8, 30, 78
 class member, 85, 445
 container defined type, 317
 member function definition, 262
 to override name lookup, 449
shift, 155, 190
sizeof, 167
subscript ([])
 bitset, 105
 deque, 325
 map, 363
 string, 87
 vector, 94, 325
and multi-dimensioned array, 142
and pointer, 124
array, 113
valid subscript range, 88
yields lvalue, 88
subtraction (-), 150
 iterator, 101, 313
 pointer, 123
unary, 148, 189
unary minus (-), 150
unary plus (+), 150
operator alternative name, 46
operator delete function, 760, 806
 compared to deallocate, 761
 compared to delete expression, 760
operator delete member, 764
 and inheritance, 764
 example, 769
 CachedObj, 770
 interface, 764
operator delete [] member, 765
operator new function, 760, 806
 compared to allocate, 761
 compared to new expression, 760
operator new member, 764
 example, 769
 CachedObj, 769
 interface, 764
operator new [] member, 765
operator overloading, *see* overloaded operator
options to main, 243
order of construction, 456, 749
 derived objects, 581, 582
 multiple base classes, 732
 virtual base classes, 746
order of destruction, 485, 749
 derived objects, 587
multiple base classes, 733
virtual base classes, 747
order of evaluation, 149, 189
 comma operator, 172
 conditional operator, 172
 logical operator, 172
ordering, strict weak, 360, 389
OrQuery, 609
 definition, 618
 eval function, 619
ostream, 6, 29, 285
 see also manipulator
 condition state, 287
 floatfield member, 831
 flushing output buffer, 290
 format state, 825
 inheritance hierarchy, 740
 no containers of, 310
 no copy or assign, 287
 not flushed if program crashes, 292
 output (<<), 7
 precedence and associativity, 158
 precision member, 829
 seek and tell members, 838
 tie member, 292
 unsetf member, 831
ostream_iterator, 407, 425
 and class type, 410
 constructors, 408
 limitations, 411
 operations, 410
 output iterator, 417
 output operator (<<), 408
 used with algorithms, 411
ostringstream, 285, 299–301
 see also ostream
 str, 301
out (file mode), 296
out_of_range, 219, 325
out_of_stock, 699
output, standard, 6
output (<<), 7, 30
 bitset, 106
 ostream_iterator, 408
 string, 81, 108
overloaded operator, 513
 formatting, 514
 must be nonmember, 514
 precedence and associativity, 158
 Sales_item, 514
output iterator, 416, 425

- overflow, 150
overflow_error, 219
 overload resolution, *see* function matching
 overloaded function, 265, 281
 using declarations, 728
 using directive, 729
 compared to redeclaration, 266
 compared to template specialization, 673
 friend declaration, 467
 linkage directive, 803
 namespaces, 727
 scope, 268
 virtual, 593
 overloaded member function, 436
 on `const`, 442
 overloaded operator, 189, 482, 503
 `<<` (output operator), 513
 `*` (dereference), 524
 `&` (address-of), 511
 `->` (arrow operator), 525
 `[]` (subscript), 522
 reference return, 522
 `()` (call operator), 530
 `=` (assignment), 476, 483, 520
 and copy constructor, 484
 check for self-assignment, 490
 Message, 490
 reference return, 483, 521
 rule of three, 485
 use counting, 495, 498
 valuelike classes, 501
 `>>` (input operator), 515
 error handling, 516–517
 must be nonmember, 514
 `<<` (output operator)
 formatting, 514
 must be nonmember, 514
 `Sales_item`, 514
 `&&` (logical AND), 511
 `||` (logical OR), 511
 `,` (comma operator), 511
 addition (`+`), `Sales_item`, 517
 ambiguous, 550
 arithmetic operators, 517
 as virtual function, 615
 binary operator, 508
 candidate functions, 549
 compound assignment (e.g., `+=`), 511
 `Sales_item`, 521
 consistency between relational and equality operators, 520
 definition, 482, 506
 design, 510–513
 equality operators, 512, 518
 explicit call to, 509
 explicit call to postfix operators, 529
 function matching, 547–551
 member and `this` pointer, 483
 member vs. nonmember function, 508, 512
 postfix increment (`++`) and decrement (`--`) operators, 528
 precedence and associativity, 507
 prefix increment (`++`) and decrement (`--`) operators, 527
 relational operators, 511, 520
 require class-type parameter, 507
 unary operator, 508
 overloading, *see* overloaded function
 operator, *see* overloaded operator
- ## P
- `pair`, 356, 388
 as return type from `map::insert`, 365
 as return type from `set::insert`, 373
 default constructor, 357
 definition, 356
 initialization, 356
 `make_pair`, 358
 operations, 357
 public data members, 357
 Panda, 731
 virtual inheritance, 741
 parameter, 226, 227, 281
 array and buffer overflow, 242
 array type, 238–244
 C-style string, 242
 `const`, 231
 `const` reference, 235
 overloading, 275
 ellipsis, 244
 function pointer, 278
 linkage directive, 804
 initialization of, 229
 iterator, 238, 242
 library container, 237
 lifetime, 255

- local copy, 230
- matching, 229
 - ellipsis, 244
 - template specialization, 673
 - with class type conversion, 541
- multi-dimensioned array, 241
- nonconst reference, 232
- nonreference type, 230
 - uses copy constructor, 478
- of member function, 260
- vector type, 237
- passing, 229
- pointer to `const`, 231
 - overloading, 275
- pointer to function, 278
 - linkage directive, 804
- pointer to `nonconst`, 231
- pointer type, 231, 239
- reference
 - to array type, 240
 - to pointer, 236
- template, *see* template parameter
- and `main`, 243
- type checking
 - and template argument, 638
 - of reference to array, 240
- parameter list, 3, 29, 226, 228
 - member function definition, 446
- parentheses, override precedence, 169
- partial specialization, 678, 684
- `partial_sort`, 818
- `partial_sort_copy`, 818
- `partial_sum`, 824
- `partition`, 817
- `placement new`, 761, 807
 - compared to `construct`, 762
- `plus<T>`, 534
- pointer, 114, 114–126, 146
 - array, 122
 - arrow (`->`), 164
 - as initializer of `vector`, 140
 - as parameter, 231
 - assignment, 120
 - `char*`, *see* C-style string
 - class member copy control, 492–501
 - copy constructor, 480
 - destructor, 485
 - strategies, 499
 - compared to iterator, 114
 - compared to reference, 121
 - `const`, 128
- const pointer to `const`, 129
- container constructor from, 308
- conversion from derived to base, 567
- conversion from derived to multiple base, 734
- conversion to `bool`, 182
- conversion to `void`, 181
- dangling, 176, 188
 - synthesized copy control, 494
- declaration style, 116–117
- definition, 115
- `delete`, 176
- `dynamic_cast`, example, 773
- function returning, 228
- implicit `this`, 260, 281
- initialization, 117–119
- is polymorphic, 569
- multi-dimensioned array, 143
- `new`, 174
- null, 118
- off-the-end, 125
- pitfalls with generic programs, 671
- reference parameter, 236
- relational operator, 132
- return type and local variable, 249
- smart, 495, 503, 553
 - handle class, 599
 - overloaded `->` (arrow operator) and `*` (dereference), 524
 - overloaded `(++)` and `(*)`, 526
- subscript operator, 124
- to pointer, 122
- `typedef`, 129
- `typeid` operator, 776
- uninitialized, 117
- `volatile`, 800
- pointer arithmetic, 123, 146
- pointer to `const`, 127
 - argument, 231
 - conversion from `nonconst`, 127
 - parameter, 231
 - overloading, 275
- pointer to function, 276–279
 - definition, 276
 - exception specifications, 711
 - function returning, 228
 - initialization, 277
 - linkage directive, 803
 - overloaded functions, 279
 - parameter, 278
 - return type, 278

typedef, 276
 pointer to member, 780, 807
 and typedef, 783
 arrow (->*), 783
 definition, 781
 dot (. *), 783
 function pointer, 782
 function table, 785
 pointer to nonconst
 argument, 231
 parameter, 231
 polymorphism, 558, 622
 compile time polymorphism via templates, 624
 run time polymorphism in C++, 569
 pop
 priority_queue, 352
 queue, 352
 stack, 351
 pop_back, sequential container, 326
 pop_front, sequential container, 326
 portable, 797
 postfix decrement (--)
 overloaded operator, 528
 yields rvalue, 163
 postfix increment (++)
 and dereference, 163
 overloaded operator, 528
 precedence, 124, 146, 149, 168, 189
 of assignment, 160
 of conditional, 166
 of dot and dereference, 164
 of increment and dereference, 163
 of IO operator, 158
 of pointer to member and call operator, 782
 overloaded operator, 507
 pointer parameter declaration, 241
 precedence table, 170
 predicate, 402, 425
 prefix decrement (--), 163
 overloaded operator, 527
 yields lvalue, 162
 prefix increment (++)
 and dereference, 163
 overloaded operator, 527
 yields lvalue, 162
 preprocessor, 70, 75
 directive, 7, 29
 macro, 221, 224
 variable, 71
 prev_permutation, 821
 preventing copies of class objects, 481
 print_total, 559
 explained, 568
 printable character, 88
 printValues program, 240, 242, 243
 priority_queue, 348, 354
 constructors, 349
 relational operator, 350
 private
 class, 496
 copy constructor, 481
 inheritance, 571
 member, 75, 474
 private access label, 65, 432
 inheritance, 561
 private inheritance, 622
 program
 factorial, 250
 find_val, 234
 gcd, 226
 isShorter, 235
 make_plural, 248
 open_file, 299
 printValues, 240, 242, 243
 ptr_swap, 237
 rgcd, 250
 swap, 233, 245
 vector capacity, 331
 book finding
 using equal_range, 379
 using find, 377
 using upper_bound, 378
 bookstore, 26
 bookstore exception classes, 698
 CachedObj, 766
 duplicate words, 400–404
 revisited, 531
 find last word, 414
 GT6, 403
 Handle class, 667
 int instantiation, 668
 operations, 668
 Sales_item instantiation, 669
 isShorter, 403
 message handling classes, 486
 Query
 design, 609–611
 interface, 610
 operations, 607
 Queue, 648

- copy_elems member, 652
destroy member, 651
pop member, 651
push member, 652
random IO example, 840
restricted word count, 374
`Sales_item` handle class, 599
Screen class template, 655
`TextQuery`, 383
 class definition, 382
 design, 380
 interface, 381
`vector`, capacity, 331
vowel counting, 200
word count, 363
word transformation, 370
`ZooAnimal` class hierarchy, 731
programmer-defined header, 67–72
programming
 generic, 95, 624
 object-oriented, 285, 302, 622
promotion, integral, 180, 189
protected, inheritance, 571, 622
protected access label, 562, 622
protected keyword, 562
prototype, function, 251, 281
`ptr_swap` program, 237
`ptrdiff_t`, 123, 146
public
 inheritance, 571, 622
 member, 75, 474
`public` access label, 65, 432
 inheritance, 561
pure virtual function, 596, 622
 example, 609
push
 `priority_queue`, 352
 queue, 352
 stack, 351
`push_back`, 94, 108
 vector, 94
 `back_inserter`, 399
 sequential container, 318
`push_front`
 `front_inserter`, 406
 sequential container, 318
`put_Msg_in_Folder`, 489
- Q**
- `Query`, 610
- & (bitwise AND), 610
 definition, 614
~ (bitwise NOT), 610
 definition, 614
| (bitwise OR), 610
 definition, 614
<< (output operator), 615
definition, 613
design, 609–611
interface, 610
operations, 607
`Query_base`, 609
 definition, 612
 member functions, 609
`Queue`
 << (output operator), 659
 assign, 662
 copy_elems member, 652, 662
 definition, 648
 design, 647
 destroy member, 651
 final class definition, 664
 interface, 627
 member template declarations, 661
 operations, 627
 pop member, 651
 push, specialized, 677
 push member, 652
 template version, `char*`, 675
`queue`, 348, 354
 constructors, 349
 relational operator, 350
`QueueItem`, 648
 as nested class, 787
 constructor, 789
 definition, 788
 friendship, 658
`CachedObj`, 768
 allocation explained, 769
- R**
- Raccoon as virtual base, 741
RAII, *see* resource allocation is initialization
raise, 750
raise exception, *see* throw
random file IO, 838
random-access iterator, 417, 425
 `deque`, 417
 `string`, 417

vector, 417
`random_shuffle`, 820
 range
 iterator, 314, 314–316, 354
 left-inclusive, 314
 range_error, 219
`rbegin`, container, 317, 412
`rdstate`, 290
 recursive function, 249, 281
 refactoring, 583, 622
 referece, 317
 reference, 58, 75
 and pointer, 121
 const reference, 59
 initialization, 60
 conversion from derived to base, 567
 conversion from derived to multiple
 base, 734
 dynamic_cast operator, example,
 774
 is polymorphic, 569
 nonconst reference, 60
 parameter, 232–237
 pointer parameter, 236
 return type, is lvalue, 249
 return type and class object, 440
 return type and local variable, 249
 return value, 248
 to array parameter, 240
 reference count, *see* use count
 reference data member, initialization, 455
 reference to const, *see* const reference
`reinterpret_cast`, 183, 185
 relational operator, 153
 string, 85
 associative container, 359
 container, 321
 container adaptor, 350
 function object, 533
 overloaded operator, 511, 520
 consistent with equality, 520
 pointer, 132
`remove`, 819
 list, 423
`remove_copy`, 820
`remove_copy_if`, 820
`remove_if`, 819
 list, 423
`remove_Msg_from_Folder`, 491
`rend`, container, 317, 412
`replace`, 400, 816
 string, 342
`replace_copy`, 400, 815
`replace_copy_if`, 815
`replace_if`, 816
`reserve`
 string, 336
 vector, 331
 reserved identifier, 47
`resize`, sequential container, 323
 Resource, 700
 resource allocation is initialization, 700–
 701
 `auto_ptr`, 702
 restricted word count program, 374
 result, 148, 189
`rethrow`, 695, 750
`return`, container, 381
`return statement`, 245–251
 from main, 247
 implicit, 245
 local variable, 247, 249
 return type, 3, 29, 226, 227
 const reference, 249
 function, 281
 function pointer, 278
 linkage directive, 804
 member function definition, 446
 no implicit return type, 228
 nonreference, 247
 uses copy constructor, 478
 of virtual function, 564
 pointer to function, 278
 reference, 248
 reference yields lvalue, 249
 void, 245
`return value`
 conversion, 246
 copied, 247
`reverse`, 819
 list, 423
 reverse iterator, 405, 412–415, 425
 -- (decrement), 412
 ++ (increment), 412
 base, 414
 compared to iterator, 413, 414
 example, 414
 requires -- (decrement), 413
`reverse_copy`, 819
`reverse_iterator`, 412
 container, 316
`rfind`, string, 346

rgcd program, 250
right manipulator, 832
right-shift (>>), 155, 190
rotate, 819
rotate_copy, 820
rule of three, 485, 503
 exception for virtual destructors, 588
run time, 75
 error, 38
run-time type identification, 772–780, 807
 classes with virtual functions, 772
 compared to virtual functions, 777
 dynamic_cast, 773
 example, 773
 throws bad_cast, 774
 to pointer, 773
 to reference, 774
 type-sensitive equality, 778
typeid, 775
 and virtual functions, 775
 example, 776
 returns type_info, 776
runtime_error, 217, 219
 constructor from string, 218
rvalue, 45, 75

S

safety, exception, 700
Sales_item, 21
 addition (+), 23, 517
 throws exception, 217, 699
 class definition, 64, 258–265
 compare function, 604
 compound assignment (e.g., +=), 521
 conversion, 461
 default constructor, 263
 equality operators (==), (!=), 519
 explicit constructor, 462
 handle class, 599
 clone function, 602
 constructor, 601, 602
 definition, 600
 design, 599
 multiset of, 605
 using generic Handle, 669
 header, 21, 67, 264
 input (>>), 516
 istream constructor, 452
 no relational operators, 520
 operations, 21

output (<<), 514
avg_price definition, 261
same_isbn, 24, 258
string constructor, 452
scientific manipulator, 830
scope, 54, 75
 const object, 57, 69
 block, 193
 class, 65, 444, 473
 compared to object lifetime, 254
 for statement, 15
 friend declaration, 466
 function, 227
 function template specialization, 674
 global, 54, 74
 local, 54, 75
 multiple inheritance, 737
 namespace, 713–717
 statement, 194
 template parameter, 629
 using declaration, 720
 using directive, 721
 example, 722
 name collisions, 723
scope (::)
 base class members, 569
 namespace member, 750
scope operator (::), 8, 30, 78, 108
 class member, 85, 445
 container defined type, 317
 member function definition, 262
 namespace member, 713
 to override class-specific memory allocation, 765
 to override name lookup, 449
Screen, 435
 class template, 655
 concatenating operations, 441
 display, 442
 do_display, 442
 friends, 465
 get definition, 446
 get members, 436
 get_cursor definition, 446
 CachedObj, 768
 Menu function table, 785
 move members, 441
 set members, 441
 simplified, 781
 size_type, 435
ScreenPtr, 523

arrow operator (->), 525
 dereference (*), 524
 use counted, 523
 ScrPtr, 523
 search, 813
 search_n, 813
 self-assignment
 auto_ptr, 705
 check, 490
 use counting, 498
 semantics, value, 499, 503
 semicolon (;), 3
 semicolon (;), class definition, 440
 sentinel, 97, 108
 separate compilation, 67, 76
 inclusion model for templates, 644
 of templates, 643
 separate compilation model for templates, 645, 684
 sequence, escape, 74
 sequence (\Xnnn), hexadecimal escape, 40
 sequential container, 306, 354
 assign, 328
 assignment (=), 328
 back, 324
 clear, 327
 const_iterator, 316
 const_reverse_iterator, 316
 constructor from element count
 uses copy constructor, 478
 uses element default constructor, 460
 constructors, 307–309
 deque, 306
 element type constraints, 309, 323
 empty, 323
 erase, 326
 front, 324
 insert, 319
 iterator, 316
 list, 306
 operations, 316–330
 performance characteristics, 333
 pop_back, 326
 pop_front, 326
 priority_queue, 348
 push_back, 318
 push_front, 318
 queue, 348
 rbegin, 412
 rend, 412
 resize, 323
 returning a, 381
 reverse_iterator, 316, 412
 size, 323
 size_type, 316
 stack, 348
 supports relational operators, 321
 swap, 329
 types defined by, 316
 value_type, 317
 vector, 306
 set, 356, 388
 as element type, 311
 assignment (=), 328
 begin, 372
 bidirectional iterator, 417
 clear, 359
 constructors, 372
 count, 372
 element type constraints, 309
 empty, 359
 end, 372
 equal_range, 379
 erase, 359, 372
 find, 372
 insert, 373
 iterator, 374
 key type constraints, 360
 lower_bound, 377
 operations, 358
 overriding the default comparison, 604
 rbegin, 412
 rend, 412
 return alternatives, 381
 return type from insert, 373
 reverse_iterator, 412
 size, 359
 supports relational operators, 359
 swap, 329
 upper_bound, 377
 value_type, 372
 set_difference, 822
 set_intersection, 619, 822
 set_symmetric_difference, 822
 set_union, 822
 setfill manipulator, 832
 setprecision manipulator, 829
 setstate, 289, 290
 setw manipulator, 832
 shift operator, 155, 190

- short, 34
short-circuit evaluation, 152
 overloaded operator, 508
shorterString, 248
showbase manipulator, 827
showpoint manipulator, 831
signed, 35, 76
 conversion to unsigned, 36, 180
size, 108
 string, 83
 vector, 93
 associative container, 359
 priority_queue, 352
 queue, 352
 sequential container, 323
 stack, 351
size_t, 104, 108, 146
 and array, 113
size_type, 84, 108
 string, 84
 vector, 93
 container, 316
sizeof operator, 167
skipws manipulator, 833
sliced, 579, 622
SmallInt, 536, 550
 conversion operator, 537
smart pointer, 495, 503, 553
 handle class, 599
 overloaded -> (arrow operator) and
 * (dereference), 524
 overloaded (++) and (*), 526
sort, 401, 817
source file, 4, 29
 naming convention, 264
specialization
 class template
 definition, 675
 member defined outside class body, 676
 partial, 678
 partial specialization, 684
 class template member, 677
 declaration, 677
function template
 compared to overloaded function, 673
 declaration, 672, 673
 example, 672
 scope, 674
template, namespaces, 730
specifier, type, 48, 76
splice, list, 423
sstream
 header, 285, 300
 str, 301
stable_partition, 817
stable_sort, 403, 817
stack, 348, 354
 constructors, 349
 relational operator, 350
stack unwinding, 691, 750
standard error, 6, 29
standard input, 6, 29
standard library, 5, 29
standard output, 6, 30
state, condition, 302
statement, 2, 30
 break, 212, 223
 continue, 214, 223
 do while, 210
 for, 29, 207
 goto, 214, 224
 if, 17, 29, 195, 224
 return, 245–251
 switch, 199, 224
 while, 12, 30, 204, 224
compound, 193, 223
declaration, 193, 224
expression, 192, 224
for statement/for, 14
labeled, 214, 224
null, 192, 224
 return, local variable, 247, 249
statement block, *see* block
statement label, 214
statement scope, 194
statement/for statement, for, 14
static (file static), 719
static class member, 467, 474
 as default argument, 471
class template, 665
 accessed through an instantiation, 666
 definition, 666
const data member, initialization, 470
const member function, 469
data member, 469
 as constant expression, 471
inheritance, 576
member function, 467

this pointer, 469
static object, local, 255, 281
static type, 568, 622
 determines name lookup, 590
 multiple inheritance, 735
static type checking, 44, 76
 argument, 229
 function return value, 246
`static_cast`, 183, 185
`std`, 8, 30
`stdexcept` header, 217, 219
`store`, free, 135, 145
`str`, 301
`strcat`, 133
`strcmp`, 133
`strcpy`, 133
stream
 `istream_iterator`, 407
 `ostream_iterator`, 407
 flushing buffer, 290
 iterator, 405, 407–412
 and class type, 410
 limitations, 411
 used with algorithms, 411
 not flushed if program crashes, 292
 type as condition, 19
stream iterator, 425
strict weak ordering, 360, 389
string, C-style, *see* C-style string
string, 80–89
 addition, 86
 addition to string literal, 87
 and string literal, 81, 140
 append, 342
 are case sensitive, 344
 as sequential container, 335
 `assign`, 340
 assignment (`=`), 86
 `c_str`, 140
 `c_str`, example, 294
 capacity, 336
 compare, 347
 compared to C-style string, 134
 compound assignment, 86
 concatenation, 86
 constructor, 80, 338–339
 default constructor, 52
 `empty`, 83
 equality (`==`), 85
 equality operator, 347
 `erase`, 340
 `find`, 344
 `find_first_not_of`, 346
 `find_first_of`, 345
 `find_last_not_of`, 346
 `find_last_of`, 346
 `getline`, 82
 `getline`, example, 300
 header, 80
 input operation as condition, 82
 input operator, 81
 `insert`, 340
 output operator, 81
 random-access iterator, 417
 relational operator, 85, 347
 `replace`, 342
 `reserve`, 336
 `rfind`, 346
 `size`, 83
 `size_type`, 84
 subscript operator, 87
 `substr`, 342
string literal, 9, 30, 40
 addition to string, 87
 and string library type, 81, 140
 and C-style string, 140
 concatenation, 41
`stringstream`, 285, 299–301, 302
 see also `istream`
 see also `ostream`
 `str`, 301
`strlen`, 133
`strncat`, 133
`strncpy`, 133
struct, *see also* class
 default access label, 433
 default inheritance access label, 574
struct, keyword, 66, 76, 474
 in variable definition, 440
structure, data, 20, 28
Studio, Visual, 4
`subscript([])`, 87, 108, 146, 389
 `bitset`, 105
 `deque`, 325
 `map`, 363
 `string`, 87
 `vector`, 94, 325
 and multi-dimensioned array, 142
 and pointer, 124
 array, 113
 overloaded operator, 522
 reference return, 522

- valid subscript range, 88
yields lvalue, 88
- subscript range
 string, 88
 vector, 96
 array, 114
- substr, string, 342
- subtraction (-), 150
 iterator, 101, 313
 pointer, 123
- swap, 329, 816
 container, 329
- swap program, 233, 245
- swap_ranges, 816
- switch statement, 199, 224
 default label, 203
 and break, 201–203
 case label, 201
 compared to if, 199
 execution flow, 201
 expression, 203
 variable definition, 204
- synthesized assignment (=), 483, 503
 multiple inheritance, 737
 pointer members, 493
- synthesized copy constructor, 479, 503
 multiple inheritance, 737
 pointer members, 493
 virtual base class, 747
- synthesized copy control, volatile, 800
- synthesized default constructor, 264, 281, 459, 474
 inheritance, 581
- synthesized destructor, 485, 486
 multiple inheritance, 737
 pointer members, 493
- T**
- \t (tab character), 40
- table of library name and header, 810
- template
 see also class template
 see also function template
 see also instantiation
 class, 90, 107
 class member, *see* member template
 link time errors, 635
 overview, 624
- template keyword, 625
- template argument, 625, 684
 and function argument type checking, 638
 class template, 628
 conversion, 638
 deduction, 684
 from function pointer, 640
 deduction for class template member function, 653
 deduction for function template, 637
 explicit and class template, 636
 explicit and function template, 642
 and function pointer, 643
 specifying, 642
 pointer, 671
- template argument deduction, 637
- template class, *see* class template
- template function, *see* function template
- template parameter, 625, 628–633, 684
 and member templates, 663
 name, 628
 restrictions on use, 629
- nontype parameter, 625, 628, 632, 684
 class template, 655
 must be constant expression, 633
- scope, 629
- type parameter, 625, 628, 630, 684
- uses of inside class definition, 649
- template parameter list, 625, 684
- template specialization, 672, 684
 class member declaration, 677
 compared to overloaded function, 673
 definition, 675
 example, 672
 function declaration, 672, 673
 member defined outside class body, 676
 member of class template, 677
 parameter matching, 673
 partial specialization, 678, 684
 scope, 674
- template<>, *see* template specialization
- temporary object, 247
- terminate, 219, 219, 224, 692, 750
- TextQuery
 class definition, 382
 main program using, 383
 program design, 380
 program interface, 381
 revisited, 609
- this pointer
 implicit, 260, 281

- implicit parameter, 431, 440
- in overloaded operator, 483
- overloaded operator, 508
- static member functions, 469
- three**, rule of, **485**, 503
- throw**, **216**, **216**, **224**, **689**, 750
 - example, 217, 290
 - execution flow, 218, 691
 - pointer to local object, 690
 - rethrow, 695
- tolower**, 88
- top**
 - priority_queue, 352
 - stack, 351
- toupper**, 88
- transform**, 815
- transformation program, word, 370
- translation unit, *see* source file
- trunc** (file mode), 296
- try block**, **216**, **217**, **224**, 750
- try keyword**, 217
- type**
 - abstract data, 78, 473
 - arithmetic, **34**, 73
 - built-in, 3, 28, 34–37
 - class, **20**, 28, 65
 - compound, **58**, 73, 145
 - dynamic, **568**, 622
 - function return, 281
 - incomplete, **438**, 474
 - integral, **34**, 75
 - library, 29
 - nested, *see* nested class**786**
 - return, 3, 29, **226**, 227
 - static, **568**, 622
 - determines name lookup, 590
 - name lookup and multiple inheritance, 735
- type checking**, 44
 - argument, 229
 - with class type conversion, 541
 - ellipsis parameter, 244
 - name lookup, 269
 - reference to array argument, 240
- type identification**, run-time, 772–780, 807
- type specifier**, **48**, 76
- type template parameter**, 628, 630, 684
 - see also* template parameter
- type_info**, 807
 - header, 219
 - name member, 780
- no copy or assign, 780
- operations, 779
- returned from typeid, 776
- typedef**
 - and pointer, 129
 - and pointer to member, 783
 - pointer to function, 276
- typedef**, **61**, 76
- typeid operator**, **775**, **807**
 - and virtual functions, 775
 - example, 776
 - returns type_info, 776
- typename**, keyword
 - compared to class, 631
 - in template parameter, 630
 - inside template definition, 632

U

- U_Ptr**, 496
- unary function object**, **533**
- unary minus (-)**, 150
- unary operator**, **148**, 189
- unary plus (+)**, 150
- uncaught exception, 692
- undefined behavior, **41**, 76
 - dangling pointer, 176
 - synthesized copy control, 494
- invalidated iterator, 315
- uninitialized class data member, 459
- uninitialized pointer, 117
- uninitialized variable, 51
- underflow_error**, 219
- unexpected**, **708**, 750
- uninitialized, **8**, **30**, **51**, 76
- uninitialized pointer, 117
- uninitialized_copy**, 755, 759
- uninitialized_fill**, 755
- union**, **792**, 807
 - anonymous, **795**, 805
 - as nested type, 794
 - example, 794
 - limitations on, 793
- union keyword**, 793
- unique**, 402, 819
 - list, 423
- unique_copy**, 412, 820
- unitbuf**, manipulator flushes the buffer, 291
- unnamed namespace, **718**, 750
 - local to file, 718

- replace file static, 719
unsigned, 35, 76
 conversion to signed, 36, 180
 literal (*numU* or *numu*), 39
unsigned char, 36
unwinding, stack, 691, 750
upper_bound, 814
 associative container, 377
 book finding program, 378
 example, 607
uppercase manipulator, 828
use count, 495, 503
 design overview, 495
 generic class, 667
 held in companion class, 496
 pointer to, 600
 self-assignment check, 498
user, 433, 563
using declaration, 78, 108, 720, 750
 access control, 573
 class member access, 574
 in header, 80
 overloaded function, 728
 overloaded inherited functions, 593
 scope, 720
using directive, 721, 751
 overloaded function, 729
pitfalls, 724
scope, 721
 example, 722
 name collisions, 723
utility header, 356
- V**
- value initialization, 92, 108
 map subscript operator, 363
 vector, 92
 and dynamically allocated array, 136
 deque~~deque~~, 309
 list~~list~~, 309
 of dynamically allocated object, 175
 and `resize`, 324
 sequential container, 309
 vector~~vector~~, 309
value semantics, 499, 503
value_type, 389
 map, multimap, 361
 sequential container, 317
 set, multiset, 372
varargs, 244
variable, 8, 30, 43–55
 define before use, 44
 defined after case label, 204
 definition, 48
 definitions and `goto`, 215
 initialization, 48, 50, 76
 constructor, 452
 local, 227, 281
 scope, 55
Vector, 757
 capacity, 757
 memory allocation strategy, 757
 push_back, 758
 reallocate, 758
 size, 757
 using operator new and delete, 761
 using explicit destructor call, 763
 using placement new, 762
vector, 90–95, 354
 argument, 237
 as element type, 311
 assign, 328
 assignment (=), 328
 at, 325
 back, 324
 begin, 97, 317
 capacity, 331
 clear, 327
 const_iterator, 99, 316
 const_reference, 317
 const_reverse_iterator, 316
 constructor from element count, uses
 copy constructor, 478
 constructor taking iterators, 140
 constructors, 91–92, 307–309
 difference_type, 316
 element type constraints, 309, 323
 empty, 93, 323
 end, 97, 317
 erase, 326, 402
 invalidates iterator, 326
 front, 324
 header, 90
 initialization from pointer, 140
 insert, 319
 invalidates iterator, 320
 iterator, 97, 316
 iterator supports arithmetic, 312
 memory allocation strategy, 756
 memory management strategy, 330

- parameter, 237
 performance characteristics, 334
`pop_back`, 326
`push_back`, 94, 318
 invalidates iterator, 321
 random-access iterator, 417
`rbegin`, 317, 412
`reference`, 317
 relational operators, 321
`rend`, 317, 412
`reserve`, 331
`resize`, 323
`reverse_iterator`, 316, 412
`size`, 93, 323
`size_type`, 93, 316
`subscript([])`, 325
 subscript operator, 94
 supports relational operators, 313
`swap`, 329
 type, 91
 types defined by, 316
`value_type`, 317
`vector` capacity program, 331
 viable function, 270, 282
 with class type conversion, 545
 virtual base class, 741, 751
 ambiguities, 743
 conversion, 743
 defining base as, 742
 derived class constructor, 744
 name lookup, 743
 order of construction, 746
 stream types, 741
 virtual function, 559, 566–570, 622
 assignment operator, 588
 calls resolved at run time, 568
 compared to run-time type identification, 777
 default argument, 570
 derived classes, 564
 destructor, 587
 multiple inheritance, 736
 exception specifications, 710
 in constructors, 589
 in destructor, 589
 introduction, 561
 multiple inheritance, 735
 no virtual constructor, 588
 overloaded, 593
 overloaded operator, 615
 overriding run-time binding, 570
 pure, 596, 622
 example, 609
 return type, 564
 run-time type identification, 772
 scope, 594
 static, 469
 to copy unknown type, 602
 type-sensitive equality, 778
 virtual inheritance, 741, 751
 virtual keyword, 559
 Visual Studio, 4
`void`, 34, 76
 return type, 245
`void*`, 119, 146
 const `void*`, 127, 145
`volatile`, 800, 807
 pointer, 800
 synthesized copy control, 800
 vowel counting program, 200
- ## W
- `wcerr`, 286
`wchar_t`, 34
 literal, 40
`wchar_t` streams, 286
`wcin`, 286
`wcout`, 286
 weak ordering, strict, 360, 389
`wfstream`, 286
`what`, *see* exception
 while statement, 12, 30, 204, 224
 condition in, 205
 whitespace, 81
 wide character streams, 286
`wifstream`, 286
`window`, console, 6
`Window_Mgr`, 465
`wiostream`, 286
`wistream`, 286
`wistringstream`, 286
`wofstream`, 286
`word`, 35, 76
 word count program, 363
 restricted, 374
 word per line processing
 `istringstream`, 386
 `istringstream`*istringstream*, 370
`istringstream`, 300
 word transformation program, 370
 WordQuery, 609

definition, 616
`wostream`, 286
`wostringstream`, 286
wrap around, 38
`wstringstream`, 286

X

`\Xnnn` (hexadecimal escape sequence), 40

Z

`ZooAnimal`, using virtual inheritance, 741
`ZooAnimal` class hierarchy, 731