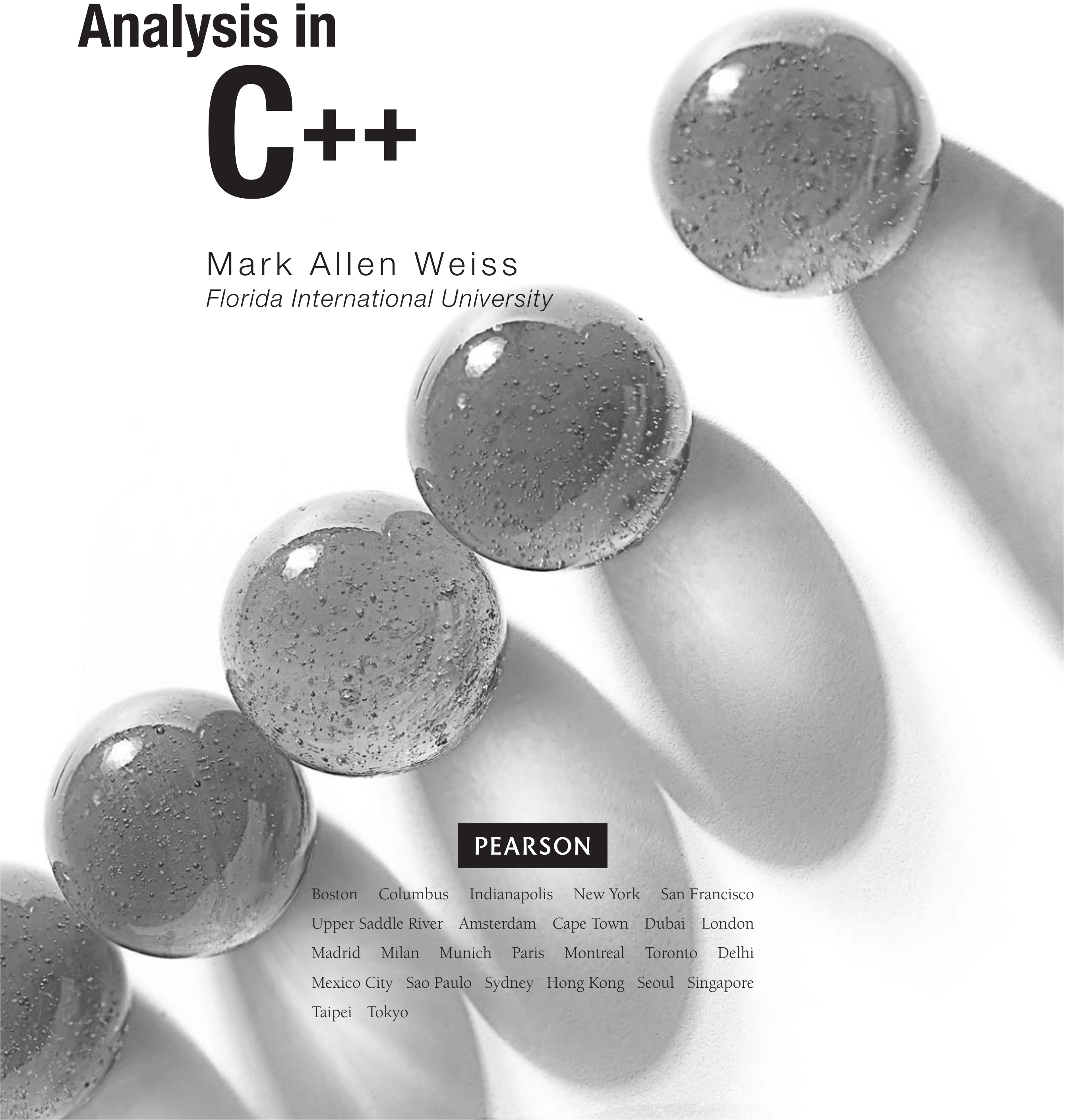
Fourth Edition

Data Structures and Algorithm Analysis in C++

Mark Allen Weiss
Florida International University



PEARSON

Boston Columbus Indianapolis New York San Francisco
Upper Saddle River Amsterdam Cape Town Dubai London
Madrid Milan Munich Paris Montreal Toronto Delhi
Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore
Taipei Tokyo

Editorial Director, ECS: Marcia Horton
Executive Editor: Tracy Johnson
Editorial Assistant: Jenah Blitz-Stoehr
Director of Marketing: Christy Lesko
Marketing Manager: Yez Alayan
Senior Marketing Coordinator: Kathryn Ferranti
Marketing Assistant: Jon Bryant
Director of Production: Erin Gregg
Senior Managing Editor: Scott Disanno
Senior Production Project Manager: Marilyn Lloyd
Manufacturing Buyer: Linda Sager
Art Director: Jayne Conte
Cover Designer: Bruce Kenselaar
Permissions Supervisor: Michael Joyce
Permissions Administrator: Jenell Forschler
Cover Image: © De-kay | Dreamstime.com
Media Project Manager: Renata Butera
Full-Service Project Management: Integra Software Services Pvt. Ltd.
Composition: Integra Software Services Pvt. Ltd.
Text and Cover Printer/Binder: Courier Westford

Library of Congress Cataloging-in-Publication Data

Weiss, Mark Allen.
Data structures and algorithm analysis in C++ / Mark Allen Weiss, Florida International University. — Fourth edition.
pages cm
ISBN-13: 978-0-13-284737-7 (alk. paper)
ISBN-10: 0-13-284737-X (alk. paper)
1. C++ (Computer program language) 2. Data structures (Computer science) 3. Computer algorithms. I. Title.
QA76.73.C153W46 2014
005.7′3—dc23
2013011064

10 9 8 7 6 5 4 3 2 1

PEARSON

www.pearsonhighered.com

ISBN-10: 0-13-284737-X
ISBN-13: 978-0-13-284737-7

To my kind, brilliant, and inspiring Sara.

This page intentionally left blank

CONTENTS

PREFACE

Purpose/Goals

The fourth edition of *Data Structures and Algorithm Analysis in C++* describes *data structures,* methods of organizing large amounts of data, and *algorithm analysis,* the estimation of the running time of algorithms. As computers become faster and faster, the need for programs that can handle large amounts of input becomes more acute. Paradoxically, this requires more careful attention to efficiency, since inefficiencies in programs become most obvious when input sizes are large. By analyzing an algorithm before it is actually coded, students can decide if a particular solution will be feasible. For example, in this text students look at specific problems and see how careful implementations can reduce the time constraint for large amounts of data from centuries to less than a second. Therefore, no algorithm or data structure is presented without an explanation of its running time. In some cases, minute details that affect the running time of the implementation are explored.

Once a solution method is determined, a program must still be written. As computers have become more powerful, the problems they must solve have become larger and more complex, requiring development of more intricate programs. The goal of this text is to teach students good programming and algorithm analysis skills simultaneously so that they can develop such programs with the maximum amount of efficiency.

This book is suitable for either an advanced data structures course or a first-year graduate course in algorithm analysis. Students should have some knowledge of intermediate programming, including such topics as pointers, recursion, and object-based programming, as well as some background in discrete math.

Approach

Although the material in this text is largely language-independent, programming requires the use of a specific language. As the title implies, we have chosen C++ for this book.

C++ has become a leading systems programming language. In addition to fixing many of the syntactic flaws of C, C++ provides direct constructs (the *class* and *template*) to implement generic data structures as abstract data types.

The most difficult part of writing this book was deciding on the amount of C++ to include. Use too many features of C++ and one gets an incomprehensible text; use too few and you have little more than a C text that supports classes.

The approach we take is to present the material in an *object-based approach.* As such, there is almost no use of inheritance in the text. We use class templates to describe generic data structures. We generally avoid esoteric C++ features and use the `vector` and `string` classes that are now part of the C++ standard. Previous editions have implemented class templates by separating the class template interface from its implementation. Although this is arguably the preferred approach, it exposes compiler problems that have made it difficult for readers to actually use the code. As a result, in this edition the online code represents class templates as a single unit, with no separation of interface and implementation. Chapter 1 provides a review of the C++ features that are used throughout the text and describes our approach to class templates. Appendix A describes how the class templates could be rewritten to use separate compilation.

Complete versions of the data structures, in both C++ and Java, are available on the Internet. We use similar coding conventions to make the parallels between the two languages more evident.

Summary of the Most Significant Changes in the Fourth Edition

The fourth edition incorporates numerous bug fixes, and many parts of the book have undergone revision to increase the clarity of presentation. In addition,

- Chapter 4 includes implementation of the AVL tree deletion algorithm—a topic often requested by readers.
- Chapter 5 has been extensively revised and enlarged and now contains material on two newer algorithms: cuckoo hashing and hopscotch hashing. Additionally, a new section on universal hashing has been added. Also new is a brief discussion of the `unordered_set` and `unordered_map` class templates introduced in C++11.
- Chapter 6 is mostly unchanged; however, the implementation of the binary heap makes use of move operations that were introduced in C++11.
- Chapter 7 now contains material on radix sort, and a new section on lower-bound proofs has been added. Sorting code makes use of move operations that were introduced in C++11.
- Chapter 8 uses the new union/find analysis by Seidel and Sharir and shows the $O(\ M\,\alpha(M,N)\ )$ bound instead of the weaker $O(\ M \log^* N\ )$ bound in prior editions.
- Chapter 12 adds material on suffix trees and suffix arrays, including the linear-time suffix array construction algorithm by Karkkainen and Sanders (with implementation). The sections covering deterministic skip lists and AA-trees have been removed.
- Throughout the text, the code has been updated to use C++11. Notably, this means use of the new C++11 features, including the `auto` keyword, the range `for` loop, move construction and assignment, and uniform initialization.

Overview

Chapter 1 contains review material on discrete math and recursion. I believe the only way to be comfortable with recursion is to see good uses over and over. Therefore, recursion is prevalent in this text, with examples in every chapter except Chapter 5. Chapter 1 also includes material that serves as a review of basic C++. Included is a discussion of templates and important constructs in C++ class design.

Chapter 2 deals with algorithm analysis. This chapter explains asymptotic analysis and its major weaknesses. Many examples are provided, including an in-depth explanation of logarithmic running time. Simple recursive programs are analyzed by intuitively converting them into iterative programs. More complicated divide-and-conquer programs are introduced, but some of the analysis (solving recurrence relations) is implicitly delayed until Chapter 7, where it is performed in detail.

Chapter 3 covers lists, stacks, and queues. This chapter includes a discussion of the STL `vector` and `list` classes, including material on iterators, and it provides implementations of a significant subset of the `STL vector and list` classes.

Chapter 4 covers trees, with an emphasis on search trees, including external search trees (B-trees). The UNIX file system and expression trees are used as examples. AVL trees and splay trees are introduced. More careful treatment of search tree implementation details is found in Chapter 12. Additional coverage of trees, such as file compression and game trees, is deferred until Chapter 10. Data structures for an external medium are considered as the final topic in several chapters. Included is a discussion of the STL `set` and `map` classes, including a significant example that illustrates the use of three separate maps to efficiently solve a problem.

Chapter 5 discusses hash tables, including the classic algorithms such as separate chaining and linear and quadratic probing, as well as several newer algorithms, namely cuckoo hashing and hopscotch hashing. Universal hashing is also discussed, and extendible hashing is covered at the end of the chapter.

Chapter 6 is about priority queues. Binary heaps are covered, and there is additional material on some of the theoretically interesting implementations of priority queues. The Fibonacci heap is discussed in Chapter 11, and the pairing heap is discussed in Chapter 12.

Chapter 7 covers sorting. It is very specific with respect to coding details and analysis. All the important general-purpose sorting algorithms are covered and compared. Four algorithms are analyzed in detail: insertion sort, Shellsort, heapsort, and quicksort. New to this edition is radix sort and lower bound proofs for selection-related problems. External sorting is covered at the end of the chapter.

Chapter 8 discusses the disjoint set algorithm with proof of the running time. This is a short and specific chapter that can be skipped if Kruskal's algorithm is not discussed.

Chapter 9 covers graph algorithms. Algorithms on graphs are interesting, not only because they frequently occur in practice but also because their running time is so heavily dependent on the proper use of data structures. Virtually all of the standard algorithms are presented along with appropriate data structures, pseudocode, and analysis of running time. To place these problems in a proper context, a short discussion on complexity theory (including *NP*-completeness and undecidability) is provided.

Chapter 10 covers algorithm design by examining common problem-solving techniques. This chapter is heavily fortified with examples. Pseudocode is used in these later chapters so that the student's appreciation of an example algorithm is not obscured by implementation details.

Chapter 11 deals with amortized analysis. Three data structures from Chapters 4 and 6 and the Fibonacci heap, introduced in this chapter, are analyzed.

Chapter 12 covers search tree algorithms, the suffix tree and array, the *k*-d tree, and the pairing heap. This chapter departs from the rest of the text by providing complete and careful implementations for the search trees and pairing heap. The material is structured so that the instructor can integrate sections into discussions from other chapters. For example, the top-down red-black tree in Chapter 12 can be discussed along with AVL trees (in Chapter 4).

Chapters 1 to 9 provide enough material for most one-semester data structures courses. If time permits, then Chapter 10 can be covered. A graduate course on algorithm analysis could cover chapters 7 to 11. The advanced data structures analyzed in Chapter 11 can easily be referred to in the earlier chapters. The discussion of *NP*-completeness in Chapter 9 is far too brief to be used in such a course. You might find it useful to use an additional work on *NP*-completeness to augment this text.

Exercises

Exercises, provided at the end of each chapter, match the order in which material is presented. The last exercises may address the chapter as a whole rather than a specific section. Difficult exercises are marked with an asterisk, and more challenging exercises have two asterisks.

References

References are placed at the end of each chapter. Generally the references either are historical, representing the original source of the material, or they represent extensions and improvements to the results given in the text. Some references represent solutions to exercises.

Supplements

The following supplements are available to all readers at http://cssupport.pearsoncmg.com/

- Source code for example programs
- Errata

In addition, the following material is available only to qualified instructors at Pearson Instructor Resource Center (www.pearsonhighered.com/irc). Visit the IRC or contact your Pearson Education sales representative for access.

- Solutions to selected exercises
- Figures from the book
- Errata

Acknowledgments

Many, many people have helped me in the preparation of books in this series. Some are listed in other versions of the book; thanks to all.

As usual, the writing process was made easier by the professionals at Pearson. I'd like to thank my editor, Tracy Johnson, and production editor, Marilyn Lloyd. My wonderful wife Jill deserves extra special thanks for everything she does.

Finally, I'd like to thank the numerous readers who have sent e-mail messages and pointed out errors or inconsistencies in earlier versions. My website www.cis.fiu.edu/~weiss will also contain updated source code (in C++ and Java), an errata list, and a link to submit bug reports.

M.A.W.
Miami, Florida

CHAPTER 1

Programming: A General Overview

In this chapter, we discuss the aims and goals of this text and briefly review programming concepts and discrete mathematics. We will . . .

- See that how a program performs for reasonably large input is just as important as its performance on moderate amounts of input.
- Summarize the basic mathematical background needed for the rest of the book.
- Briefly review **recursion**.
- Summarize some important features of C++ that are used throughout the text.

1.1 What's This Book About?

Suppose you have a group of N numbers and would like to determine the kth largest. This is known as the **selection problem**. Most students who have had a programming course or two would have no difficulty writing a program to solve this problem. There are quite a few "obvious" solutions.

One way to solve this problem would be to read the N numbers into an array, sort the array in decreasing order by some simple algorithm such as bubble sort, and then return the element in position k.

A somewhat better algorithm might be to read the first k elements into an array and sort them (in decreasing order). Next, each remaining element is read one by one. As a new element arrives, it is ignored if it is smaller than the kth element in the array. Otherwise, it is placed in its correct spot in the array, bumping one element out of the array. When the algorithm ends, the element in the kth position is returned as the answer.

Both algorithms are simple to code, and you are encouraged to do so. The natural questions, then, are: Which algorithm is better? And, more important, Is either algorithm good enough? A simulation using a random file of 30 million elements and $k = 15{,}000{,}000$ will show that neither algorithm finishes in a reasonable amount of time; each requires several days of computer processing to terminate (albeit eventually with a correct answer). An alternative method, discussed in Chapter 7, gives a solution in about a second. Thus, although our proposed algorithms work, they cannot be considered good algorithms,

	1	2	3	4
1	t	h	i	s
2	w	a	t	s
3	o	a	h	g
4	f	g	d	t

Figure 1.1 Sample word puzzle

because they are entirely impractical for input sizes that a third algorithm can handle in a reasonable amount of time.

A second problem is to solve a popular word puzzle. The input consists of a two-dimensional array of letters and a list of words. The object is to find the words in the puzzle. These words may be horizontal, vertical, or diagonal in any direction. As an example, the puzzle shown in Figure 1.1 contains the words *this, two, fat,* and *that.* The word *this* begins at row 1, column 1, or (1,1), and extends to (1,4); *two* goes from (1,1) to (3,1); *fat* goes from (4,1) to (2,3); and *that* goes from (4,4) to (1,1).

Again, there are at least two straightforward algorithms that solve the problem. For each word in the word list, we check each ordered triple (*row, column, orientation*) for the presence of the word. This amounts to lots of nested `for` loops but is basically straightforward.

Alternatively, for each ordered quadruple (*row, column, orientation, number of characters*) that doesn't run off an end of the puzzle, we can test whether the word indicated is in the word list. Again, this amounts to lots of nested `for` loops. It is possible to save some time if the maximum number of characters in any word is known.

It is relatively easy to code up either method of solution and solve many of the real-life puzzles commonly published in magazines. These typically have 16 rows, 16 columns, and 40 or so words. Suppose, however, we consider the variation where only the puzzle board is given and the word list is essentially an English dictionary. Both of the solutions proposed require considerable time to solve this problem and therefore might not be acceptable. However, it is possible, even with a large word list, to solve the problem very quickly.

An important concept is that, in many problems, writing a working program is not good enough. If the program is to be run on a large data set, then the running time becomes an issue. Throughout this book we will see how to estimate the running time of a program for large inputs and, more important, how to compare the running times of two programs without actually coding them. We will see techniques for drastically improving the speed of a program and for determining program bottlenecks. These techniques will enable us to find the section of the code on which to concentrate our optimization efforts.

1.2 Mathematics Review

This section lists some of the basic formulas you need to memorize, or be able to derive, and reviews basic proof techniques.

1.2.1 Exponents

$$X^A X^B = X^{A+B}$$

$$\frac{X^A}{X^B} = X^{A-B}$$

$$(X^A)^B = X^{AB}$$

$$X^N + X^N = 2X^N \neq X^{2N}$$

$$2^N + 2^N = 2^{N+1}$$

1.2.2 Logarithms

In computer science, all logarithms are to the base 2 unless specified otherwise.

Definition 1.1
$X^A = B$ if and only if $\log_X B = A$

Several convenient equalities follow from this definition.

Theorem 1.1

$$\log_A B = \frac{\log_C B}{\log_C A}; \quad A, B, C > 0, A \neq 1$$

Proof
Let $X = \log_C B$, $Y = \log_C A$, and $Z = \log_A B$. Then, by the definition of logarithms, $C^X = B$, $C^Y = A$, and $A^Z = B$. Combining these three equalities yields $B = C^X = (C^Y)^Z$. Therefore, $X = YZ$, which implies $Z = X/Y$, proving the theorem.

Theorem 1.2

$$\log AB = \log A + \log B; \quad A, B > 0$$

Proof
Let $X = \log A$, $Y = \log B$, and $Z = \log AB$. Then, assuming the default base of 2, $2^X = A$, $2^Y = B$, and $2^Z = AB$. Combining the last three equalities yields $2^X 2^Y = AB = 2^Z$. Therefore, $X + Y = Z$, which proves the theorem.

Some other useful formulas, which can all be derived in a similar manner, follow.

$$\log A/B = \log A - \log B$$

$$\log(A^B) = B \log A$$

$$\log X < X \quad \text{for all } X > 0$$

$$\log 1 = 0, \qquad \log 2 = 1, \qquad \log 1{,}024 = 10, \qquad \log 1{,}048{,}576 = 20$$

1.2.3 Series

The easiest formulas to remember are

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

and the companion,

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

In the latter formula, if $0 < A < 1$, then

$$\sum_{i=0}^{N} A^i \le \frac{1}{1 - A}$$

and as N tends to ∞, the sum approaches $1/(1 - A)$. These are the "geometric series" formulas.

We can derive the last formula for $\sum_{i=0}^{\infty} A^i$ $(0 < A < 1)$ in the following manner. Let S be the sum. Then

$$S = 1 + A + A^2 + A^3 + A^4 + A^5 + \cdots$$

Then

$$AS = A + A^2 + A^3 + A^4 + A^5 + \cdots$$

If we subtract these two equations (which is permissible only for a convergent series), virtually all the terms on the right side cancel, leaving

$$S - AS = 1$$

which implies that

$$S = \frac{1}{1 - A}$$

We can use this same technique to compute $\sum_{i=1}^{\infty} i/2^i$, a sum that occurs frequently. We write

$$S = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + \cdots$$

and multiply by 2, obtaining

$$2S = 1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \frac{5}{2^4} + \frac{6}{2^5} + \cdots$$

Subtracting these two equations yields

$$S = 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \cdots$$

Thus, $S = 2$.

Another type of common series in analysis is the arithmetic series. Any such series can be evaluated from the basic formula:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

For instance, to find the sum $2 + 5 + 8 + \cdots + (3k - 1)$, rewrite it as $3(1 + 2 + 3 + \cdots + k) - (1 + 1 + 1 + \cdots + 1)$, which is clearly $3k(k+1)/2 - k$. Another way to remember this is to add the first and last terms (total $3k + 1$), the second and next-to-last terms (total $3k + 1$), and so on. Since there are $k/2$ of these pairs, the total sum is $k(3k + 1)/2$, which is the same answer as before.

The next two formulas pop up now and then but are fairly uncommon.

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

$$\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|} \qquad k \neq -1$$

When $k = -1$, the latter formula is not valid. We then need the following formula, which is used far more in computer science than in other mathematical disciplines. The numbers H_N are known as the harmonic numbers, and the sum is known as a harmonic sum. The error in the following approximation tends to $\gamma \approx 0.57721566$, which is known as **Euler's constant**.

$$H_N = \sum_{i=1}^{N} \frac{1}{i} \approx \log_e N$$

These two formulas are just general algebraic manipulations:

$$\sum_{i=1}^{N} f(N) = Nf(N)$$

$$\sum_{i=n_0}^{N} f(i) = \sum_{i=1}^{N} f(i) - \sum_{i=1}^{n_0-1} f(i)$$

1.2.4 Modular Arithmetic

We say that A is congruent to B modulo N, written $A \equiv B \pmod{N}$, if N divides $A - B$. Intuitively, this means that the remainder is the same when either A or B is divided by N. Thus, $81 \equiv 61 \equiv 1 \pmod{10}$. As with equality, if $A \equiv B \pmod{N}$, then $A + C \equiv B + C \pmod{N}$ and $AD \equiv BD \pmod{N}$.

Often, N is a prime number. In that case, there are three important theorems:

First, if N is prime, then $ab \equiv 0 \pmod{N}$ is true if and only if $a \equiv 0 \pmod{N}$ or $b \equiv 0 \pmod{N}$. In other words, if a prime number N divides a product of two numbers, it divides at least one of the two numbers.

Second, if N is prime, then the equation $ax \equiv 1 \pmod{N}$ has a unique solution $\pmod{N}$ for all $0 < a < N$. This solution, $0 < x < N$, is the *multiplicative inverse*.

Third, if N is prime, then the equation $x^2 \equiv a \pmod{N}$ has either two solutions $\pmod{N}$ for all $0 < a < N$, or it has no solutions.

There are many theorems that apply to modular arithmetic, and some of them require extraordinary proofs in number theory. We will use modular arithmetic sparingly, and the preceding theorems will suffice.

1.2.5 The *P* Word

The two most common ways of proving statements in data-structure analysis are proof by induction and proof by contradiction (and occasionally proof by intimidation, used by professors only). The best way of proving that a theorem is false is by exhibiting a counterexample.

Proof by Induction

A proof by induction has two standard parts. The first step is proving a *base case,* that is, establishing that a theorem is true for some small (usually degenerate) value(s); this step is almost always trivial. Next, an **inductive hypothesis** is assumed. Generally this means that the theorem is assumed to be true for all cases up to some limit k. Using this assumption, the theorem is then shown to be true for the next value, which is typically $k + 1$. This proves the theorem (as long as k is finite).

As an example, we prove that the Fibonacci numbers, $F_0 = 1, F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, \ldots, F_i = F_{i-1} + F_{i-2}$, satisfy $F_i < (5/3)^i$, for $i \geq 1$. (Some definitions have $F_0 = 0$, which shifts the series.) To do this, we first verify that the theorem is true for the trivial cases. It is easy to verify that $F_1 = 1 < 5/3$ and $F_2 = 2 < 25/9$; this proves the basis. We assume that the theorem is true for $i = 1, 2, \ldots, k$; this is the inductive hypothesis. To prove the theorem, we need to show that $F_{k+1} < (5/3)^{k+1}$. We have

$$F_{k+1} = F_k + F_{k-1}$$

by the definition, and we can use the inductive hypothesis on the right-hand side, obtaining

$$\begin{aligned} F_{k+1} &< (5/3)^k + (5/3)^{k-1} \\ &< (3/5)(5/3)^{k+1} + (3/5)^2(5/3)^{k+1} \\ &< (3/5)(5/3)^{k+1} + (9/25)(5/3)^{k+1} \end{aligned}$$

which simplifies to

$$\begin{aligned} F_{k+1} &< (3/5 + 9/25)(5/3)^{k+1} \\ &< (24/25)(5/3)^{k+1} \\ &< (5/3)^{k+1} \end{aligned}$$

proving the theorem.

As a second example, we establish the following theorem.

Theorem 1.3
If $N \geq 1$, then $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$

Proof
The proof is by induction. For the basis, it is readily seen that the theorem is true when $N = 1$. For the inductive hypothesis, assume that the theorem is true for $1 \leq k \leq N$. We will establish that, under this assumption, the theorem is true for $N + 1$. We have

$$\sum_{i=1}^{N+1} i^2 = \sum_{i=1}^{N} i^2 + (N+1)^2$$

Applying the inductive hypothesis, we obtain

$$\begin{aligned} \sum_{i=1}^{N+1} i^2 &= \frac{N(N+1)(2N+1)}{6} + (N+1)^2 \\ &= (N+1)\left[\frac{N(2N+1)}{6} + (N+1)\right] \\ &= (N+1)\frac{2N^2 + 7N + 6}{6} \\ &= \frac{(N+1)(N+2)(2N+3)}{6} \end{aligned}$$

Thus,

$$\sum_{i=1}^{N+1} i^2 = \frac{(N+1)[(N+1)+1][2(N+1)+1]}{6}$$

proving the theorem.

Proof by Counterexample

The statement $F_k \leq k^2$ is false. The easiest way to prove this is to compute $F_{11} = 144 > 11^2$.

Proof by Contradiction

Proof by contradiction proceeds by assuming that the theorem is false and showing that this assumption implies that some known property is false, and hence the original assumption was erroneous. A classic example is the proof that there is an infinite number of primes. To prove this, we assume that the theorem is false, so that there is some largest prime P_k. Let $P_1, P_2, \ldots, P_k$ be all the primes in order and consider

$$N = P_1P_2P_3 \cdots P_k + 1$$

Clearly, N is larger than P_k, so, by assumption, N is not prime. However, none of $P_1, P_2, \ldots, P_k$ divides N exactly, because there will always be a remainder of 1. This is a contradiction, because every number is either prime or a product of primes. Hence, the original assumption, that P_k is the largest prime, is false, which implies that the theorem is true.

1.3 A Brief Introduction to Recursion

Most mathematical functions that we are familiar with are described by a simple formula. For instance, we can convert temperatures from Fahrenheit to Celsius by applying the formula

$$C = 5(F - 32)/9$$

Given this formula, it is trivial to write a C++ function; with declarations and braces removed, the one-line formula translates to one line of C++.

Mathematical functions are sometimes defined in a less standard form. As an example, we can define a function f, valid on nonnegative integers, that satisfies $f(0) = 0$ and $f(x) = 2f(x - 1) + x^2$. From this definition we see that $f(1) = 1, f(2) = 6, f(3) = 21$, and $f(4) = 58$. A function that is defined in terms of itself is called **recursive**. C++ allows functions to be recursive.[1] It is important to remember that what C++ provides is merely an attempt to follow the recursive spirit. Not all mathematically recursive functions are efficiently (or correctly) implemented by C++'s simulation of recursion. The idea is that the recursive function f ought to be expressible in only a few lines, just like a nonrecursive function. Figure 1.2 shows the recursive implementation of f.

Lines 3 and 4 handle what is known as the **base case**, that is, the value for which the function is directly known without resorting to recursion. Just as declaring $f(x) = 2f(x - 1) + x^2$ is meaningless, mathematically, without including the fact that $f(0) = 0$, the recursive C++ function doesn't make sense without a base case. Line 6 makes the recursive call.

```
1   int f( int x )
2   {
3       if( x == 0 )
4           return 0;
5       else
6           return 2 * f( x - 1 ) +  x * x;
7   }
```

Figure 1.2 A recursive function

[1] Using recursion for numerical calculations is usually a bad idea. We have done so to illustrate the basic points.

There are several important and possibly confusing points about recursion. A common question is: Isn't this just circular logic? The answer is that although we are defining a function in terms of itself, we are not defining a particular instance of the function in terms of itself. In other words, evaluating $f(5)$ by computing $f(5)$ would be circular. Evaluating $f(5)$ by computing $f(4)$ is not circular—unless, of course, $f(4)$ is evaluated by eventually computing $f(5)$. The two most important issues are probably the *how* and *why* questions. In Chapter 3, the *how* and *why* issues are formally resolved. We will give an incomplete description here.

It turns out that recursive calls are handled no differently from any others. If f is called with the value of 4, then line 6 requires the computation of $2 * f(3) + 4 * 4$. Thus, a call is made to compute $f(3)$. This requires the computation of $2*f(2)+3*3$. Therefore, another call is made to compute $f(2)$. This means that $2 * f(1) + 2 * 2$ must be evaluated. To do so, $f(1)$ is computed as $2*f(0)+1*1$. Now, $f(0)$ must be evaluated. Since this is a base case, we know a priori that $f(0) = 0$. This enables the completion of the calculation for $f(1)$, which is now seen to be 1. Then $f(2)$, $f(3)$, and finally $f(4)$ can be determined. All the bookkeeping needed to keep track of pending function calls (those started but waiting for a recursive call to complete), along with their variables, is done by the computer automatically. An important point, however, is that recursive calls will keep on being made until a base case is reached. For instance, an attempt to evaluate $f(-1)$ will result in calls to $f(-2)$, $f(-3)$, and so on. Since this will never get to a base case, the program won't be able to compute the answer (which is undefined anyway). Occasionally, a much more subtle error is made, which is exhibited in Figure 1.3. The error in Figure 1.3 is that `bad(1)` is defined, by line 6, to be `bad(1)`. Obviously, this doesn't give any clue as to what `bad(1)` actually is. The computer will thus repeatedly make calls to `bad(1)` in an attempt to resolve its values. Eventually, its bookkeeping system will run out of space, and the program will terminate abnormally. Generally, we would say that this function doesn't work for one special case but is correct otherwise. This isn't true here, since `bad(2)` calls `bad(1)`. Thus, `bad(2)` cannot be evaluated either. Furthermore, `bad(3)`, `bad(4)`, and `bad(5)` all make calls to `bad(2)`. Since `bad(2)` is not evaluable, none of these values are either. In fact, this program doesn't work for any nonnegative value of `n`, except 0. With recursive programs, there is no such thing as a "special case."

These considerations lead to the first two fundamental rules of recursion:

1. *Base cases*. You must always have some base cases, which can be solved without recursion.
2. *Making progress*. For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case.

```
1  int bad( int n )
2  {
3      if( n == 0 )
4          return 0;
5      else
6          return bad( n / 3 + 1 ) + n - 1;
7  }
```

Figure 1.3 A nonterminating recursive function

Throughout this book, we will use recursion to solve problems. As an example of a nonmathematical use, consider a large dictionary. Words in dictionaries are defined in terms of other words. When we look up a word, we might not always understand the definition, so we might have to look up words in the definition. Likewise, we might not understand some of those, so we might have to continue this search for a while. Because the dictionary is finite, eventually either (1) we will come to a point where we understand all of the words in some definition (and thus understand that definition and retrace our path through the other definitions) or (2) we will find that the definitions are circular and we are stuck, or that some word we need to understand for a definition is not in the dictionary.

Our recursive strategy to understand words is as follows: If we know the meaning of a word, then we are done; otherwise, we look the word up in the dictionary. If we understand all the words in the definition, we are done; otherwise, we figure out what the definition means by *recursively* looking up the words we don't know. This procedure will terminate if the dictionary is well defined but can loop indefinitely if a word is either not defined or circularly defined.

Printing Out Numbers

Suppose we have a positive integer, n, that we wish to print out. Our routine will have the heading `printOut(n)`. Assume that the only I/O routines available will take a single-digit number and output it. We will call this routine `printDigit`; for example, `printDigit(4)` will output a 4.

Recursion provides a very clean solution to this problem. To print out 76234, we need to first print out 7623 and then print out 4. The second step is easily accomplished with the statement `printDigit(n%10)`, but the first doesn't seem any simpler than the original problem. Indeed it is virtually the same problem, so we can solve it recursively with the statement `printOut(n/10)`.

This tells us how to solve the general problem, but we still need to make sure that the program doesn't loop indefinitely. Since we haven't defined a base case yet, it is clear that we still have something to do. Our base case will be `printDigit(n)` if $0 \leq n < 10$. Now `printOut(n)` is defined for every positive number from 0 to 9, and larger numbers are defined in terms of a smaller positive number. Thus, there is no cycle. The entire function is shown in Figure 1.4.

We have made no effort to do this efficiently. We could have avoided using the mod routine (which can be very expensive) because $n\%10 = n - \lfloor n/10 \rfloor * 10$ is true for positive n.[2]

```
void printOut( int n )  // Print nonnegative n
{
    if( n >= 10 )
        printOut( n / 10 );
    printDigit( n % 10 );
}
```

Figure 1.4 Recursive routine to print an integer

[2] $\lfloor x \rfloor$ is the largest integer that is less than or equal to x.

Recursion and Induction

Let us prove (somewhat) rigorously that the recursive number-printing program works. To do so, we'll use a proof by induction.

Theorem 1.4
The recursive number-printing algorithm is correct for $n \geq 0$.

Proof *(By induction on the number of digits in* n*)*
First, if n has one digit, then the program is trivially correct, since it merely makes a call to `printDigit`. Assume then that `printOut` works for all numbers of k or fewer digits. A number of $k+1$ digits is expressed by its first k digits followed by its least significant digit. But the number formed by the first k digits is exactly $\lfloor n/10 \rfloor$, which, by the inductive hypothesis, is correctly printed, and the last digit is $n \mod 10$, so the program prints out any $(k+1)$-digit number correctly. Thus, by induction, all numbers are correctly printed.

This proof probably seems a little strange in that it is virtually identical to the algorithm description. It illustrates that in designing a recursive program, all smaller instances of the same problem (which are on the path to a base case) may be *assumed* to work correctly. The recursive program needs only to combine solutions to smaller problems, which are "magically" obtained by recursion, into a solution for the current problem. The mathematical justification for this is proof by induction. This gives the third rule of recursion:

3. *Design rule*. Assume that all the recursive calls work.

This rule is important because it means that when designing recursive programs, you generally don't need to know the details of the bookkeeping arrangements, and you don't have to try to trace through the myriad of recursive calls. Frequently, it is extremely difficult to track down the actual sequence of recursive calls. Of course, in many cases this is an indication of a good use of recursion, since the computer is being allowed to work out the complicated details.

The main problem with recursion is the hidden bookkeeping costs. Although these costs are almost always justifiable, because recursive programs not only simplify the algorithm design but also tend to give cleaner code, recursion should not be used as a substitute for a simple `for` loop. We'll discuss the overhead involved in recursion in more detail in Section 3.6.

When writing recursive routines, it is crucial to keep in mind the four basic rules of recursion:

1. *Base cases*. You must always have some base cases, which can be solved without recursion.
2. *Making progress*. For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case.
3. *Design rule*. Assume that all the recursive calls work.
4. *Compound interest rule*. Never duplicate work by solving the same instance of a problem in separate recursive calls.

The fourth rule, which will be justified (along with its nickname) in later sections, is the reason that it is generally a bad idea to use recursion to evaluate simple mathematical functions, such as the Fibonacci numbers. As long as you keep these rules in mind, recursive programming should be straightforward.

1.4 C++ Classes

In this text, we will write many data structures. All of the data structures will be objects that store data (usually a collection of identically typed items) and will provide functions that manipulate the collection. In C++ (and other languages), this is accomplished by using a *class*. This section describes the C++ class.

1.4.1 Basic `class` Syntax

A class in C++ consists of its **members**. These members can be either data or functions. The functions are called **member functions**. Each instance of a class is an *object*. Each object contains the data components specified in the class (unless the data components are `static`, a detail that can be safely ignored for now). A member function is used to act on an object. Often member functions are called **methods**.

As an example, Figure 1.5 is the `IntCell` class. In the `IntCell` class, each instance of the `IntCell`—an `IntCell` object—contains a single data member named `storedValue`. Everything else in this particular class is a method. In our example, there are four methods. Two of these methods are `read` and `write`. The other two are special methods known as constructors. Let us describe some key features.

First, notice the two labels `public` and `private`. These labels determine visibility of class members. In this example, everything except the `storedValue` data member is `public`. `storedValue` is `private`. A member that is `public` may be accessed by any method in any class. A member that is `private` may only be accessed by methods in its class. Typically, data members are declared `private`, thus restricting access to internal details of the class, while methods intended for general use are made `public`. This is known as **information hiding**. By using `private` data members, we can change the internal representation of the object without having an effect on other parts of the program that use the object. This is because the object is accessed through the `public` member functions, whose viewable behavior remains unchanged. The users of the class do not need to know internal details of how the class is implemented. In many cases, having this access leads to trouble. For instance, in a class that stores dates using month, day, and year, by making the month, day, and year `private`, we prohibit an outsider from setting these data members to illegal dates, such as Feb 29, 2013. However, some methods may be for internal use and can be `private`. In a class, all members are `private` by default, so the initial `public` is not optional.

Second, we see two **constructors**. A constructor is a method that describes how an instance of the class is constructed. If no constructor is explicitly defined, one that initializes the data members using language defaults is automatically generated. The `IntCell` class defines two constructors. The first is called if no parameter is specified. The second is called if an `int` parameter is provided, and uses that `int` to initialize the `storedValue` member.

```
/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    /**
     * Construct the IntCell.
     * Initial value is 0.
     */
    IntCell( )
      { storedValue = 0; }

    /**
     * Construct the IntCell.
     * Initial value is initialValue.
     */
    IntCell( int initialValue )
      { storedValue = initialValue; }

    /**
     * Return the stored value.
     */
    int read( )
      { return storedValue; }

    /**
     * Change the stored value to x.
     */
    void write( int x )
      { storedValue = x; }

  private:
    int storedValue;
};
```

Figure 1.5 A complete declaration of an `IntCell` class

1.4.2 Extra Constructor Syntax and Accessors

Although the class works as written, there is some extra syntax that makes for better code. Four changes are shown in Figure 1.6 (we omit comments for brevity). The differences are as follows:

Default Parameters

The `IntCell` constructor illustrates the **default parameter.** As a result, there are still two `IntCell` constructors defined. One accepts an `initialValue`. The other is the zero-parameter constructor, which is implied because the one-parameter constructor says that `initialValue` is optional. The default value of 0 signifies that 0 is used if no parameter is provided. Default parameters can be used in any function, but they are most commonly used in constructors.

Initialization List

The `IntCell` constructor uses an **initialization list** (Figure 1.6, line 8) prior to the body of the constructor. The initialization list is used to initialize the data members directly. In Figure 1.6, there's hardly a difference, but using initialization lists instead of an assignment statement in the body saves time in the case where the data members are class types that have complex initializations. In some cases it is required. For instance, if a data member is `const` (meaning that it is not changeable after the object has been constructed), then the data member's value can only be initialized in the initialization list. Also, if a data member is itself a class type that does not have a zero-parameter constructor, then it must be initialized in the initialization list.

Line 8 in Figure 1.6 uses the syntax

```
: storedValue{ initialValue }  { }
```

instead of the traditional

```
: storedValue( initialValue )  { }
```

The use of braces instead of parentheses is new in C++11 and is part of a larger effort to provide a uniform syntax for initialization everywhere. Generally speaking, anywhere you can initialize, you can do so by enclosing initializations in braces (though there is one important exception, in Section 1.4.4, relating to vectors).

```
 1  /**
 2   * A class for simulating an integer memory cell.
 3   */
 4  class IntCell
 5  {
 6    public:
 7      explicit IntCell( int initialValue = 0 )
 8        : storedValue{ initialValue } { }
 9      int read( ) const
10        { return storedValue; }
11      void write( int x )
12        { storedValue = x; }
13
14    private:
15      int storedValue;
16  };
```

Figure 1.6 `IntCell` class with revisions

`explicit` *Constructor*

The `IntCell` constructor is `explicit`. You should make all one-parameter constructors `explicit` to avoid behind-the-scenes type conversions. Otherwise, there are somewhat lenient rules that will allow type conversions without explicit casting operations. Usually, this is unwanted behavior that destroys strong typing and can lead to hard-to-find bugs. As an example, consider the following:

```
IntCell obj;     // obj is an IntCell
obj = 37;        // Should not compile: type mismatch
```

The code fragment above constructs an `IntCell` object `obj` and then performs an assignment statement. But the assignment statement should not work, because the right-hand side of the assignment operator is not another `IntCell`. `obj`'s `write` method should have been used instead. However, C++ has lenient rules. Normally, a one-parameter constructor defines an **implicit type conversion**, in which a temporary object is created that makes an assignment (or parameter to a function) compatible. In this case, the compiler would attempt to convert

```
obj = 37;        // Should not compile: type mismatch
```

into

```
IntCell temporary = 37;
obj = temporary;
```

Notice that the construction of the temporary can be performed by using the one-parameter constructor. The use of `explicit` means that a one-parameter constructor cannot be used to generate an implicit temporary. Thus, since `IntCell`'s constructor is declared `explicit`, the compiler will correctly complain that there is a type mismatch.

Constant Member Function

A member function that examines but does not change the state of its object is an **accessor**. A member function that changes the state is a **mutator** (because it mutates the state of the object). In the typical collection class, for instance, `isEmpty` is an accessor, while `makeEmpty` is a mutator.

In C++, we can mark each member function as being an accessor or a mutator. Doing so is an important part of the design process and should not be viewed as simply a comment. Indeed, there are important semantic consequences. For instance, mutators cannot be applied to constant objects. By default, all member functions are mutators. To make a member function an accessor, we must add the keyword `const` after the closing parenthesis that ends the parameter type list. The const-ness is part of the signature. `const` can be used with many different meanings. The function declaration can have `const` in three different contexts. Only the `const` after a closing parenthesis signifies an accessor. Other uses are described in Sections 1.5.3 and 1.5.4.

In the `IntCell` class, `read` is clearly an accessor: it does not change the state of the `IntCell`. Thus it is made a constant member function at line 9. If a member function is marked as an accessor but has an implementation that changes the value of any data member, a compiler error is generated.[3]

1.4.3 Separation of Interface and Implementation

The class in Figure 1.6 contains all the correct syntactic constructs. However, in C++ it is more common to separate the class interface from its implementation. The interface lists the class and its members (data and functions). The implementation provides implementations of the functions.

Figure 1.7 shows the class interface for `IntCell`, Figure 1.8 shows the implementation, and Figure 1.9 shows a `main` routine that uses the `IntCell`. Some important points follow.

Preprocessor Commands

The interface is typically placed in a file that ends with `.h`. Source code that requires knowledge of the interface must `#include` the interface file. In our case, this is both the implementation file and the file that contains `main`. Occasionally, a complicated project will have files including other files, and there is the danger that an interface might be read twice in the course of compiling a file. This can be illegal. To guard against this, each header file uses the preprocessor to define a symbol when the class interface is read. This is shown on the first two lines in Figure 1.7. The symbol name, `IntCell_H`, should not appear in any other file; usually, we construct it from the filename. The first line of the interface file

```
#ifndef IntCell_H
#define IntCell_H

/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 );
    int read( ) const;
    void write( int x );

  private:
    int storedValue;
};

#endif
```

Figure 1.7 `IntCell` class interface in file *IntCell.h*

[3] Data members can be marked `mutable` to indicate that const-ness should not apply to them.

```
#include "IntCell.h"

/**
 * Construct the IntCell with initialValue
 */
IntCell::IntCell( int initialValue ) : storedValue{ initialValue }
{
}

/**
 * Return the stored value.
 */
int IntCell::read( ) const
{
    return storedValue;
}

/**
 * Store x.
 */
void IntCell::write( int x )
{
    storedValue = x;
}
```

Figure 1.8 `IntCell` class implementation in file *IntCell.cpp*

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main( )
{
    IntCell m;

    m.write( 5 );
    cout << "Cell contents: " << m.read( ) << endl;

    return 0;
}
```

Figure 1.9 Program that uses `IntCell` in file *TestIntCell.cpp*

tests whether the symbol is undefined. If so, we can process the file. Otherwise, we do not process the file (by skipping to the `#endif`), because we know that we have already read the file.

Scope Resolution Operator

In the implementation file, which typically ends in `.cpp`, `.cc`, or `.C`, each member function must identify the class that it is part of. Otherwise, it would be assumed that the function is in global scope (and zillions of errors would result). The syntax is `ClassName::member`. The `::` is called the **scope resolution operator**.

Signatures Must Match Exactly

The signature of an implemented member function must match exactly the signature listed in the class interface. Recall that whether a member function is an accessor (via the `const` at the end) or a mutator is part of the signature. Thus an error would result if, for example, the `const` was omitted from exactly one of the `read` signatures in Figures 1.7 and 1.8. Note that default parameters are specified in the interface only. They are omitted in the implementation.

Objects Are Declared Like Primitive Types

In classic C++, an object is declared just like a primitive type. Thus the following are legal declarations of an `IntCell` object:

```
IntCell obj1;        // Zero parameter constructor
IntCell obj2( 12 );  // One parameter constructor
```

On the other hand, the following are incorrect:

```
IntCell obj3 = 37;   // Constructor is explicit
IntCell obj4( );     // Function declaration
```

The declaration of `obj3` is illegal because the one-parameter constructor is `explicit`. It would be legal otherwise. (In other words, in classic C++ a declaration that uses the one-parameter constructor must use the parentheses to signify the initial value.) The declaration for `obj4` states that it is a function (defined elsewhere) that takes no parameters and returns an `IntCell`.

The confusion of `obj4` is one reason for the uniform initialization syntax using braces. It was ugly that initializing with zero parameter in a constructor initialization list (Fig. 1.6, line 8) would require parentheses with no parameter, but the same syntax would be illegal elsewhere (for `obj4`). In C++11, we can instead write:

```
IntCell obj1;        // Zero parameter constructor, same as before
IntCell obj2{ 12 };  // One parameter constructor, same as before
IntCell obj4{ };     // Zero parameter constructor
```

The declaration of `obj4` is nicer because initialization with a zero-parameter constructor is no longer a special syntax case; the initialization style is uniform.

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

int main( )
{
    vector<int> squares( 100 );

    for( int i = 0; i < squares.size( ); ++i )
        squares[ i ] = i * i;

    for( int i = 0; i < squares.size( ); ++i )
        cout << i << " " << squares[ i ] << endl;

    return 0;
}
```

Figure 1.10 Using the `vector` class: stores 100 squares and outputs them

1.4.4 `vector` and `string`

The C++ standard defines two classes: the `vector` and `string`. `vector` is intended to replace the built-in C++ array, which causes no end of trouble. The problem with the built-in C++ array is that it does not behave like a first-class object. For instance, built-in arrays cannot be copied with `=`, a built-in array does not remember how many items it can store, and its indexing operator does not check that the index is valid. The built-in string is simply an array of characters, and thus has the liabilities of arrays plus a few more. For instance, `==` does not correctly compare two built-in strings.

The `vector` and `string` classes in the STL treat arrays and strings as first-class objects. A `vector` knows how large it is. Two `string` objects can be compared with `==`, `<`, and so on. Both `vector` and `string` can be copied with `=`. If possible, you should avoid using the built-in C++ array and string. We discuss the built-in array in Chapter 3 in the context of showing how `vector` can be implemented.

`vector` and `string` are easy to use. The code in Figure 1.10 creates a `vector` that stores one hundred perfect squares and outputs them. Notice also that `size` is a method that returns the size of the `vector`. A nice feature of the `vector` that we explore in Chapter 3 is that it is easy to change its size. In many cases, the initial size is 0 and the `vector` grows as needed.

C++ has long allowed initialization of built-in C++ arrays:

```
int daysInMonth[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

It was annoying that this syntax was not legal for `vector`s. In older C++, `vector`s were either initialized with size 0 or possibly by specifying a size. So, for instance, we would write:

```
vector<int> daysInMonth( 12 );  // No {} before C++11
daysInMonth[ 0 ] = 31; daysInMonth[ 1 ] = 28; daysInMonth[ 2 ] = 31;
daysInMonth[ 3 ] = 30; daysInMonth[ 4 ] = 31; daysInMonth[ 5 ] = 30;
daysInMonth[ 6 ] = 31; daysInMonth[ 7 ] = 31; daysInMonth[ 8 ] = 30;
daysInMonth[ 9 ] = 31; daysInMonth[ 10 ] = 30; daysInMonth[ 11 ] = 31;
```

Certainly this leaves something to be desired. C++11 fixes this problem and allows:

```
vector<int> daysInMonth = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Requiring the `=` in the initialization violates the spirit of uniform initialization, since now we would have to remember when it would be appropriate to use `=`. Consequently, C++11 also allows (and some prefer):

```
vector<int> daysInMonth { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

With syntax, however, comes ambiguity, as one sees with the declaration

```
vector<int> daysInMonth { 12 };
```

Is this a `vector` of size 12, or is it a `vector` of size 1 with a single element 12 in position 0? C++11 gives precedence to the initializer list, so in fact this is a `vector` of size 1 with a single element 12 in position 0, and if the intention is to initialize a `vector` of size 12, the old C++ syntax using parentheses must be used:

```
vector<int> daysInMonth( 12 );   // Must use () to call constructor that takes size
```

`string` is also easy to use and has all the relational and equality operators to compare the states of two strings. Thus `str1==str2` is `true` if the value of the strings are the same. It also has a `length` method that returns the string length.

As Figure 1.10 shows, the basic operation on arrays is indexing with []. Thus, the sum of the squares can be computed as:

```
int sum = 0;
for( int i = 0; i < squares.size( ); ++i )
    sum += squares[ i ];
```

The pattern of accessing every element sequentially in a collection such as an array or a `vector` is fundamental, and using array indexing for this purpose does not clearly express the idiom. C++11 adds a **range** `for` syntax for this purpose. The above fragment can be written instead as:

```
int sum = 0;
for( int x : squares )
    sum += x;
```

In many cases, the declaration of the type in the range for statement is unneeded; if `squares` is a `vector<int>`, it is obvious that `x` is intended to be an `int`. Thus C++11 also allows the use of the reserved word `auto` to signify that the compiler will automatically infer the appropriate type:

```
int sum = 0;
for( auto x : squares )
    sum += x;
```

The range `for` loop is appropriate only if every item is being accessed sequentially and only if the index is not needed. Thus, in Figure 1.10 the two loops cannot be rewritten as range `for` loops, because the index `i` is also being used for other purposes. The range `for` loop as shown so far allows only the viewing of items; changing the items can be done using syntax described in Section 1.5.4.

1.5 C++ Details

Like any language, C++ has its share of details and language features. Some of these are discussed in this section.

1.5.1 Pointers

A **pointer variable** is a variable that stores the address where another object resides. It is the fundamental mechanism used in many data structures. For instance, to store a list of items, we could use a contiguous array, but insertion into the middle of the contiguous array requires relocation of many items. Rather than store the collection in an array, it is common to store each item in a separate, noncontiguous piece of memory, which is allocated as the program runs. Along with each object is a link to the next object. This link is a pointer variable, because it stores a memory location of another object. This is the classic linked list that is discussed in more detail in Chapter 3.

To illustrate the operations that apply to pointers, we rewrite Figure 1.9 to dynamically allocate the `IntCell`. It must be emphasized that for a simple `IntCell` class, there is no good reason to write the C++ code this way. We do it only to illustrate dynamic memory allocation in a simple context. Later in the text, we will see more complicated classes, where this technique is useful and necessary. The new version is shown in Figure 1.11.

Declaration

Line 3 illustrates the declaration of `m`. The `*` indicates that `m` is a pointer variable; it is allowed to point at an `IntCell` object. The **value** of `m` is the address of the object that it points at.

```
 1  int main( )
 2  {
 3      IntCell *m;
 4
 5      m = new IntCell{ 0 };
 6      m->write( 5 );
 7      cout << "Cell contents: " << m->read( ) << endl;
 8
 9      delete m;
10      return 0;
11  }
```

Figure 1.11 Program that uses pointers to `IntCell` (there is no compelling reason to do this)

m is uninitialized at this point. In C++, no such check is performed to verify that m is assigned a value prior to being used (however, several vendors make products that do additional checks, including this one). The use of uninitialized pointers typically crashes programs, because they result in access of memory locations that do not exist. In general, it is a good idea to provide an initial value, either by combining lines 3 and 5, or by initializing m to the nullptr pointer.

Dynamic Object Creation

Line 5 illustrates how objects can be created dynamically. In C++ new returns a pointer to the newly created object. In C++ there are several ways to create an object using its zero-parameter constructor. The following would be legal:

```
m = new IntCell( );    // OK
m = new IntCell{ };    // C++11
m = new IntCell;       // Preferred in this text
```

We generally use the last form because of the problem illustrated by obj4 in Section 1.4.3.

Garbage Collection and delete

In some languages, when an object is no longer referenced, it is subject to automatic garbage collection; the programmer does not have to worry about it. C++ does not have garbage collection. When an object that *is allocated by* new is no longer referenced, the delete operation must be applied to the object (through a pointer). Otherwise, the memory that it consumes is lost (until the program terminates). This is known as a **memory leak**. Memory leaks are, unfortunately, common occurrences in many C++ programs. Fortunately, many sources of memory leaks can be automatically removed with care. One important rule is to not use new when an **automatic variable** can be used instead. In the original program, the IntCell is not allocated by new but instead is allocated as a local variable. In that case, the memory for the IntCell is automatically reclaimed when the function in which it is declared returns. The delete operator is illustrated at line 9 of Figure 1.11.

Assignment and Comparison of Pointers

Assignment and comparison of pointer variables in C++ is based on the value of the pointer, meaning the memory address that it stores. Thus two pointer variables are equal if they point at the same object. If they point at different objects, the pointer variables are not equal, even if the objects being pointed at are themselves equal. If lhs and rhs are pointer variables (of compatible types), then lhs=rhs makes lhs point at the same object that rhs points at.[4]

Accessing Members of an Object through a Pointer

If a pointer variable points at a class type, then a (visible) member of the object being pointed at can be accessed via the -> operator. This is illustrated at lines 6 and 7 of Figure 1.11.

[4] Throughout this text, we use lhs and rhs to signify *left-hand side* and *right-hand side* of a binary operator.

Address-of Operator (&)

One important operator is the **address-of operator** `&`. This operator returns the memory location where an object resides and is useful for implementing an alias test that is discussed in Section 1.5.6.

1.5.2 Lvalues, Rvalues, and References

In addition to pointer types, C++ defines reference types. One of the major changes in C++11 is the creation of a new reference type, known as an rvalue reference. In order to discuss rvalue references, and the more standard lvalue reference, we need to discuss the concept of lvalues and rvalues. Note that the precise rules are complex, and we provide a general description rather than focusing on the corner cases that are important in a language specification and for compiler writers.

An **lvalue** is an expression that identifies a non-temporary object. An **rvalue** is an expression that identifies a temporary object or is a value (such as a literal constant) not associated with any object.

As examples, consider the following:

```
vector<string> arr( 3 );
const int x = 2;
int y;
  ...
int z = x + y;
string str = "foo";
vector<string> *ptr = &arr;
```

With these declarations, `arr`, `str`, `arr[x]`, `&x`, `y`, `z`, `ptr`, `*ptr`, `(*ptr)[x]` are all lvalues. Additionally, `x` is also an lvalue, although it is not a modifiable lvalue. As a general rule, if you have a name for a variable, it is an lvalue, regardless of whether it is modifiable.

For the above declarations `2`, `"foo"`, `x+y`, `str.substr(0,1)` are all rvalues. `2` and `"foo"` are rvalues because they are literals. Intuitively, `x+y` is an rvalue because its value is temporary; it is certainly not `x` or `y`, but it is stored somewhere prior to being assigned to `z`. Similar logic applies for `str.substr(0,1)`.

Notice the consequence that there are some cases in which the result of a function call or operator call can be an lvalue (since `*ptr` and `arr[x]` generate lvalues) as does `cin>>x>>y` and others where it can be an rvalue; hence, the language syntax allows a function call or operator overload to specify this in the return type, and this aspect is discussed in Section 1.5.4. Intuitively, if the function call computes an expression whose value does not exist prior to the call and does not exist once the call is finished unless it is copied somewhere, it is likely to be an rvalue.

A reference type allows us to define a new name for an existing value. In classic C++, a reference can generally only be a name for an lvalue, since having a reference to a temporary would lead to the ability to access an object that has theoretically been declared as no longer needed, and thus may have had its resources reclaimed for another object. However, in C++11, we can have two types of references: lvalue references and rvalue references.

In C++11, an **lvalue reference** is declared by placing an `&` after some type. An lvalue reference then becomes a synonym (i.e., another name) for the object it references. For instance,

```
string str = "hell";
string & rstr = str;              // rstr is another name for str
rstr += 'o';                      // changes str to "hello"
bool cond = (&str == &rstr);      // true; str and rstr are same object
string & bad1 = "hello";          // illegal: "hello" is not a modifiable lvalue
string & bad2 = str + "";         // illegal: str+"" is not an lvalue
string & sub = str.substr( 0, 4 ); // illegal: str.substr( 0, 4 ) is not an lvalue
```

In C++11, an **rvalue reference** is declared by placing an `&&` after some type. An rvalue reference has the same characteristics as an lvalue reference except that, unlike an lvalue reference, an rvalue reference can also reference an rvalue (i.e., a temporary). For instance,

```
string str = "hell";
string && bad1 = "hello";          // Legal
string && bad2 = str + "";         // Legal
string && sub = str.substr( 0, 4 ); // Legal
```

Whereas lvalue references have several clear uses in C++, the utility of rvalue references is not obvious. Several uses of lvalue references will be discussed now; rvalue references are deferred until Section 1.5.3.

lvalue references use #1: aliasing complicated names

The simplest use, which we will see in Chapter 5, is to use a local reference variable solely for the purpose of renaming an object that is known by a complicated expression. The code we will see is similar to the following:

```
auto & whichList = theLists[ myhash( x, theLists.size( ) ) ];
if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
    return false;
whichList.push_back( x );
```

A reference variable is used so that the considerably more complex expression `theLists[myhash(x,theLists.size())]` does not have to be written (and then evaluated) four times. Simply writing

```
auto whichList = theLists[ myhash( x, theLists.size( ) ) ];
```

would not work; it would create a copy, and then the `push_back` operation on the last line would be applied to the copy, not the original.

lvalue references use #2: range `for` *loops*

A second use is in the range `for` statement. Suppose we would like to increment by 1 all values in a `vector`. This is easy with a `for` loop:

```
for( int i = 0; i < arr.size( ); ++i )
    ++arr[ i ];
```

But of course, a range `for` loop would be more elegant. Unfortunately, the natural code does not work, because `x` assumes a copy of each value in the `vector`.

```
for( auto x : arr )   // broken
    ++x;
```

What we really want is for `x` to be another name for each value in the `vector`, which is easy to do if `x` is a reference:

```
for( auto & x : arr ) // works
     ++x;
```

lvalue references use #3: avoiding a copy

Suppose we have a function `findMax` that returns the largest value in a `vector` or other large collection. Then given a `vector arr`, if we invoke `findMax`, we would naturally write

```
auto x = findMax( arr );
```

However, notice that if the `vector` stores large objects, then the result is that `x` will be a copy of the largest value in `arr`. If we need a copy for some reason, that is fine; however, in many instances, we only need the value and will not make any changes to `x`. In such a case, it would be more efficient to declare that `x` is another name for the largest value in `arr`, and hence we would declare `x` to be a reference (`auto` will deduce constness; if `auto` is not used, then typically a non-modifiable reference is explicitly stated with `const`):

```
auto & x = findMax( arr );
```

Normally, this means that `findMax` would also specify a return type that indicates a reference variable (Section 1.5.4).

This code illustrates two important concepts:

1. Reference variables are often used to avoid copying objects across function-call boundaries (either in the function call or the function return).
2. Syntax is needed in function declarations and returns to enable the passing and returning using references instead of copies.

1.5.3 Parameter Passing

Many languages, C and Java included, pass all parameters using **call-by-value:** the actual argument is copied into the formal parameter. However, parameters in C++ could be large complex objects for which copying is inefficient. Additionally, sometimes it is desirable to be able to alter the value being passed in. As a result of this, C++ has historically had three different ways to pass parameters, and C++11 has added a fourth. We will begin by describing the three parameter-passing mechanisms in classic C++ and then explain the new parameter-passing mechanism that has been recently added.

To see the reasons why call-by-value is not sufficient as the only parameter-passing mechanism in C++, consider the three function declarations below:

```
double average( double a, double b );    // returns average of a and b
void swap( double a, double b );         // swaps a and b; wrong parameter types
string randomItem( vector<string> arr ); // returns a random item in arr; inefficient
```

`average` illustrates an ideal use of call-by-value. If we make a call

```
double z = average( x, y );
```

then call-by-value copies `x` into `a`, `y` into `b`, and then executes the code for the `average` function definition that is fully specified elsewhere. Presuming that `x` and `y` are local variables inaccessible to `average`, it is guaranteed that when `average` returns, `x` and `y` are unchanged, which is a very desirable property. However, this desirable property is exactly why call-by-value cannot work for `swap`. If we make a call

```
swap( x, y );
```

then call-by-value guarantees that regardless of how `swap` is implemented, `x` and `y` will remain unchanged. What we need instead is to declare that `a` and `b` are references:

```
void swap( double & a, double & b );     // swaps a and b; correct parameter types
```

With this signature, `a` is a synonym for `x`, and `b` is a synonym for `y`. Changes to `a` and `b` in the implementation of `swap` are thus changes to `x` and `y`. This form of parameter passing has always been known as **call-by-reference** in C++. In C++11, this is more technically **call-by-lvalue-reference**, but we will use call-by-reference throughout this text to refer to this style of parameter passing.

The second problem with call-by-value is illustrated in `randomItem`. This function intends to return a random item from the `vector arr`; in principle, this is a quick operation consisting of the generation of a "random" number between 0 and `arr.size()-1`, inclusive, in order to determine an array index and the returning of the item at this randomly chosen array index. But using call-by-value as the parameter-passing mechanism forces the copy of the `vector vec` in the call `randomItem(vec)`. This is a tremendously expensive operation compared to the cost of computing and returning a randomly chosen array index and is completely unnecessary. Normally, the only reason to make a copy is to make changes to the copy while preserving the original. But `randomItem` doesn't intend to make any changes at all; it is just viewing `arr`. Thus, we can avoid the copy but achieve the same semantics by declaring that `arr` is a constant reference to `vec`; as a result, `arr` is a synonym for `vec`, with no copy, but since it is a `const`, it cannot be modified. This essentially provides the same viewable behavior as call-by-value. The signature would be

```
string randomItem( const vector<string> & arr ); // returns a random item in arr
```

This form of parameter passing is known as **call-by-reference-to-a-constant** in C++, but as that is overly verbose and the `const` precedes the `&`, it is also known by the simpler terminology of **call-by-constant reference.**

The parameter-passing mechanism for C++ prior to C++11 can thus generally be decided by a two-part test:

1. If the formal parameter should be able to change the value of the actual argument, then you *must use call-by-reference*.
2. Otherwise, the value of the actual argument cannot be changed by the formal parameter. If the type is a primitive type, use call-by-value. Otherwise, the type is a class type and is generally passed using call-by-constant-reference, unless it is an unusually small and easily copyable type (e.g., a type that stores two or fewer primitive types).

Put another way,

1. Call-by-value is appropriate for small objects that should not be altered by the function.
2. Call-by-constant-reference is appropriate for large objects that should not be altered by the function and are expensive to copy.
3. Call-by-reference is appropriate for all objects that may be altered by the function.

Because C++11 adds rvalue reference, there is a fourth way to pass parameters: **call-by-rvalue-reference**. The central concept is that since an rvalue stores a temporary that is about to be destroyed, an expression such as `x=rval` (where `rval` is an rvalue) can be implemented by a move instead of a copy; often moving an object's state is much easier than copying it, as it may involve just a simple pointer change. What we see here is that `x=y` can be a copy if `y` is an lvalue, but a move if `y` is an rvalue. This gives a primary use case of overloading a function based on whether a parameter is an lvalue or rvalue, such as:

```
string randomItem( const vector<string> & arr );  // returns random item in lvalue arr
string randomItem( vector<string> && arr );       // returns random item in rvalue arr

vector<string> v { "hello", "world" };
cout << randomItem( v ) << endl;                     // invokes lvalue method
cout << randomItem( { "hello", "world" } ) << endl; // invokes rvalue method
```

It is easy to test that with both functions written, the second overload is called on rvalues, while the first overload is called on lvalues, as shown above. The most common use of this idiom is in defining the behavior of `=` and in writing constructors, and this discussion is deferred until Section 1.5.6.

1.5.4 Return Passing

In C++, there are several different mechanisms for returning from a function. The most straightforward mechanism to use is **return-by-value**, as shown in these signatures:

```
double average( double a, double b );                 // returns average of a and b
LargeType randomItem( const vector<LargeType> & arr );   // potentially inefficient
vector<int> partialSum( const vector<int> & arr );       // efficient in C++11
```

These signatures all convey the basic idea that the function returns an object of an appropriate type that can be used by the caller; in all cases the result of the function call is an rvalue. However, the call to `randomItem` has potential inefficiencies. The call to `partialSum` similarly has potential inefficiencies, though in C++11 the call is likely to be very efficient.

```
 1  LargeType randomItem1( const vector<LargeType> & arr )
 2  {
 3      return arr[ randomInt( 0, arr.size( ) - 1 ) ];
 4  }
 5
 6  const LargeType & randomItem2( const vector<LargeType> & arr )
 7  {
 8      return arr[ randomInt( 0, arr.size( ) - 1 ) ];
 9  }
10
11      vector<LargeType> vec;
12        ...
13      LargeType item1 = randomItem1( vec );           // copy
14      LargeType item2 = randomItem2( vec );           // copy
15      const LargeType & item3 = randomItem2( vec );   // no copy
```

Figure 1.12 Two versions to obtain a random item in an array; second version avoids creation of a temporary `LargeType` object, but only if caller accesses it with a constant reference

First, consider two implementations of `randomItem`. The first implementation, shown in lines 1–4 of Figure 1.12 uses return-by-value. As a result, the `LargeType` at the random array index will be copied as part of the return sequence. This copy is done because, in general, return expressions could be rvalues (e.g., return `x+4`) and hence will not logically exist by the time the function call returns at line 13. But in this case, the return type is an lvalue that will exist long after the function call returns, since `arr` is the same as `vec`. The second implementation shown at lines 6–9 takes advantage of this and uses **return-by-constant-reference** to avoid an immediate copy. However, the caller must also use a constant reference to access the return value, as shown at line 15; otherwise, there will still be a copy. The constant reference signifies that we do not want to allow changes to be made by the caller by using the return value; in this case it is needed since `arr` itself is a non-modifiable `vector`. An alternative is to use `auto &` at line 15 to declare `item3`.

Figure 1.13 illustrates a similar situation in which call-by-value was inefficient in classic C++ due to the creation and eventual cleanup of a copy. Historically, C++ programmers have gone to great extent to rewrite their code in an unnatural way, using techniques involving pointers or additional parameters that decrease readability and maintainability, eventually leading to programming errors. In C++11, objects can define move semantics that can be employed when return-by-value is seen; in effect, the result `vector` will be moved to `sums`, and the `vector` implementation is optimized to allow this to be done with little more than a pointer change. This means that `partialSum` as shown in Figure 1.13 can be expected to avoid unnecessary copying and not need any changes. The details on how move semantics are implemented are discussed in Section 1.5.6; a `vector` implementation is discussed in Section 3.4. Notice that the move semantics can be called on `result` at line 9 in Figure 1.13 but not on the returned expression at line 3 in Figure 1.12. This is a consequence of the distinction between a temporary and a non-temporary, and the distinction between an lvalue reference and an rvalue reference.

```
vector<int> partialSum( const vector<int> & arr )
{
    vector<int> result( arr.size( ) );

    result[ 0 ] = arr[ 0 ];
    for( int i = 1; i < arr.size( ); ++i )
        result[ i ] = result[ i - 1 ] + arr[ i ];

    return result;
}

    vector<int> vec;
     ...
    vector<int> sums = partialSum( vec ); // Copy in old C++; move in C++11
```

Figure 1.13 Returning of a stack-allocated rvalue in C++11

In addition to the return-by-value and return-by-constant-reference idioms, functions can use **return-by-reference**. This idiom is used in a few places to allow the caller of a function to have modifiable access to the internal data representation of a class. Return-by-reference in this context is discussed in Section 1.7.2 when we implement a simple matrix class.

1.5.5 `std::swap` and `std::move`

Throughout this section, we have discussed instances in which C++11 allows the programmer to easily replace expensive copies with moves. Yet another example of this is the implementation of a `swap` routine. Swapping `doubles` is easily implemented with three copies, as shown in Figure 1.14. However, although the same logic works to swap larger types, it comes with a significant cost: Now the copies are very expensive! However, it is easy to see that there is no need to copy; what we actually want is to do moves instead of copies. In C++11, if the right-hand side of the assignment operator (or constructor) is an rvalue, then if the object supports moving, we can automatically avoid copies. In other words, if `vector<string>` supports efficient moving, and if at line 10 `x` were an rvalue, then `x` could be moved into `tmp`; similarly, if y was an rvalue at line 11, then it could be moved in to `y`. `vector` does indeed support moving; however, `x, y`, and `tmp` are all lvalues at lines 10, 11, 12 (remember, if an object has a name, it is an lvalue). Figure 1.15 shows how this problem is solved; an implementation of `swap` at lines 1–6 shows that we can use a cast to treat the right-hand side of lines 10–12 as rvalues. The syntax of a static cast is daunting; fortunately, function `std::move` exists that converts any lvalue (or rvalue) into an rvalue. Note that the name is misleading; `std::move` doesn't move anything; rather, it makes a value subject to be moved. Use of `std::move` is also shown in a revised implementation of `swap` at lines 8–13 of Figure 1.15. The swap function `std::swap` is also part of the Standard Library and will work for any type.

```
void swap( double & x, double & y )
{
    double tmp = x;
    x = y;
    y = tmp;
}

void swap( vector<string> & x, vector<string> & y )
{
    vector<string> tmp = x;
    x = y;
    y = tmp;
}
```

Figure 1.14 Swapping by three copies

```
void swap( vector<string> & x, vector<string> & y )
{
    vector<string> tmp = static_cast<vector<string> &&>( x );
    x = static_cast<vector<string> &&>( y );
    y = static_cast<vector<string> &&>( tmp );
}

void swap( vector<string> & x, vector<string> & y )
{
    vector<string> tmp = std::move( x );
    x = std::move( y );
    y = std::move( tmp );
}
```

Figure 1.15 Swapping by three moves; first with a type cast, second using `std::move`

1.5.6 The Big-Five: Destructor, Copy Constructor, Move Constructor, Copy Assignment `operator=`, Move Assignment `operator=`

In C++11, classes come with five special functions that are already written for you. These are the **destructor**, **copy constructor**, **move constructor**, **copy assignment operator**, and **move assignment operator**. Collectively these are the **big-five**. In many cases, you can accept the default behavior provided by the compiler for the big-five. Sometimes you cannot.

Destructor

The destructor is called whenever an object goes out of scope or is subjected to a `delete`. Typically, the only responsibility of the destructor is to free up any resources that were acquired during the use of the object. This includes calling `delete` for any corresponding `news`, closing any files that were opened, and so on. The default simply applies the destructor on each data member.

Copy Constructor and Move Constructor

There are two special constructors that are required to construct a new object, initialized to the same state as another object of the same type. These are the copy constructor if the existing object is an lvalue, and the move constructor if the existing object is an rvalue (i.e., a temporary that is about to be destroyed anyway). For any object, such as an `IntCell` object, a copy constructor or move constructor is called in the following instances:

- a declaration with initialization, such as

  ```
  IntCell B = C;   // Copy construct if C is lvalue; Move construct if C is rvalue
  IntCell B { C }; // Copy construct if C is lvalue; Move construct if C is rvalue
  ```

  but not

  ```
  B = C;           // Assignment operator, discussed later
  ```

- an object passed using call-by-value (instead of by `&` or `const &`), which, as mentioned earlier, should rarely be done anyway.
- an object returned by value (instead of by `&` or `const &`). Again, a copy constructor is invoked if the object being returned is an lvalue, and a move constructor is invoked if the object being returned is an rvalue.

By default, the copy constructor is implemented by applying copy constructors to each data member in turn. For data members that are primitive types (for instance, `int`, `double`, or pointers), simple assignment is done. This would be the case for the `storedValue` data member in our `IntCell` class. For data members that are themselves class objects, the copy constructor or move constructor, as appropriate, for each data member's class is applied to that data member.

Copy Assignment and Move Assignment (`operator=`)

The assignment operator is called when `=` is applied to two objects that have both been previously constructed. `lhs=rhs` is intended to copy the state of `rhs` into `lhs`. If `rhs` is an lvalue, this is done by using the copy assignment operator; if `rhs` is an rvalue (i.e., a temporary that is about to be destroyed anyway), this is done by using the move assignment operator. By default, the copy assignment operator is implemented by applying the copy assignment operator to each data member in turn.

Defaults

If we examine the `IntCell` class, we see that the defaults are perfectly acceptable, so we do not have to do anything. This is often the case. If a class consists of data members that are exclusively primitive types and objects for which the defaults make sense, the class defaults will usually make sense. Thus a class whose data members are `int`, `double`, `vector<int>`, `string`, and even `vector<string>` can accept the defaults.

The main problem occurs in a class that contains a data member that is a pointer. We will describe the problem and solutions in detail in Chapter 3; for now, we can sketch the problem. Suppose the class contains a single data member that is a pointer. This pointer points at a dynamically allocated object. The default destructor does nothing to data members that are pointers (for good reason—recall that we must `delete` ourselves). Furthermore, the copy constructor and copy assignment operator both copy the value of the pointer rather than the objects being pointed at. Thus, we will have two class instances that contain pointers that point to the same object. This is a so-called **shallow copy**. Typically, we would expect a **deep copy**, in which a clone of the entire object is made. Thus, as a result, when a class contains pointers as data members, and deep semantics are important, we typically must implement the destructor, copy assignment, and copy constructors ourselves. Doing so removes the move defaults, so we also must implement move assignment and the move constructor. As a general rule, either you accept the default for all five operations, or you should declare all five, and explicitly define, default (use the keyword `default`), or disallow each (use the keyword `delete`). Generally we will define all five.

For `IntCell`, the signatures of these operations are

```
~IntCell( );                                           // Destructor
IntCell( const IntCell & rhs );                        // Copy constructor
IntCell( IntCell && rhs );                             // Move constructor
IntCell & operator= ( const IntCell & rhs );           // Copy assignment
IntCell & operator= ( IntCell && rhs );                // Move assignment
```

The return type of `operator=` is a reference to the invoking object, so as to allow chained assignments `a=b=c`. Though it would seem that the return type should be a const reference, so as to disallow nonsense such as `(a=b)=c`, that expression is in fact allowed in C++ even for integer types. Hence, the reference return type (rather than the const reference return type) is customarily used but is not strictly required by the language specification.

If you write any of the big-five, it would be good practice to explicitly consider all the others, as the defaults may be invalid or inappropriate. In a simple example in which debugging code is placed in the destructor, no default move operations will be generated. And although unspecified copy operations are generated, that guarantee is deprecated and might not be in a future version of the language. Thus, it is best to explicitly list the copy-and-move operations again:

```
~IntCell( ) { cout << "Invoking destructor" << endl; }    // Destructor
IntCell( const IntCell & rhs ) = default;                 // Copy constructor
IntCell( IntCell && rhs ) = default;                      // Move constructor
IntCell & operator= ( const IntCell & rhs ) = default;    // Copy assignment
IntCell & operator= ( IntCell && rhs ) = default;         // Move assignment
```

Alternatively, we could disallow all copying and moving of `IntCell`s

```
IntCell( const IntCell & rhs ) = delete;                  // No Copy constructor
IntCell( IntCell && rhs ) = delete;                       // No Move constructor
IntCell & operator= ( const IntCell & rhs ) = delete;     // No Copy assignment
IntCell & operator= ( IntCell && rhs ) = delete;          // No Move assignment
```

If the defaults make sense in the routines we write, we will always accept them. However, if the defaults do not make sense, we will need to implement the destructor, copy-and-move constructors, and copy-and-move assignment operators. When the default does not work, the copy assignment operator can generally be implemented by creating a copy using the copy constructor and then swapping it with the existing object. The move assignment operator can generally be implemented by swapping member by member.

When the Defaults Do Not Work

The most common situation in which the defaults do not work occurs when a data member is a pointer type and the pointer is allocated by some object member function (such as the constructor). As an example, suppose we implement the `IntCell` by dynamically allocating an `int`, as shown in Figure 1.16. For simplicity, we do not separate the interface and implementation.

There are now numerous problems that are exposed in Figure 1.17. First, the output is three 4s, even though logically only `a` should be 4. The problem is that the default copy assignment operator and copy constructor copy the pointer `storedValue`. Thus `a.storedValue`, `b.storedValue`, and `c.storedValue` all point at the same `int` value. These copies are shallow; the pointers rather than the pointees are copied. A second, less obvious problem is a memory leak. The `int` initially allocated by `a`'s constructor remains allocated and needs to be reclaimed. The `int` allocated by `c`'s constructor is no longer referenced by any pointer variable. It also needs to be reclaimed, but we no longer have a pointer to it.

To fix these problems, we implement the big-five. The result (again without separation of interface and implementation) is shown in Figure 1.18. As we can see, once the destructor is implemented, shallow copying would lead to a programming error: Two `IntCell` objects would have `storedValue` pointing at the same `int` object. Once the first `IntCell` object's destructor was invoked to reclaim the object that its `storedValue` pointer was viewing, the second `IntCell` object would have a stale `storedValue` pointer. This is why C++11 has deprecated the prior behavior that allowed default copy operations even if a destructor was written.

```
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 )
      { storedValue = new int{ initialValue }; }

    int read( ) const
      { return *storedValue; }
    void write( int x )
      { *storedValue = x; }

  private:
    int *storedValue;
};
```

Figure 1.16 Data member is a pointer; defaults are no good

```
int f( )
{
    IntCell a{ 2 };
    IntCell b = a;
    IntCell c;

    c = b;
    a.write( 4 );
    cout << a.read( ) << endl << b.read( ) << endl << c.read( ) << endl;

    return 0;
}
```

Figure 1.17 Simple function that exposes problems in Figure 1.16

The copy assignment operator at lines 16–21 uses a standard idiom of checking for aliasing at line 18 (i.e., a self-assignment, in which the client is making a call `obj=obj`) and then copying each data field in turn as needed. On completion, it returns a reference to itself using `*this`. In C++11, copy assignment is often written using a **copy-and-swap idiom**, leading to an alternate implementation:

```
16      IntCell & operator= ( const IntCell & rhs )          // Copy assignment
17      {
18          IntCell copy = rhs;
19          std::swap( *this, copy );
20          return *this;
21      }
```

Line 18 places a copy of `rhs` into `copy` using the copy constructor. Then this `copy` is swapped into `*this`, placing the old contents into `copy`. On return, a destructor is invoked for `copy`, cleaning up the old memory. For `IntCell` this is a bit inefficient, but for other types, especially those with many complex interacting data members, it can be a reasonably good default. Notice that if `swap` were implemented using the basic copy algorithm in Figure 1.14, the copy-and-swap idiom would not work, because there would be mutual nonterminating recursion. In C++11 we have a basic expectation that swapping is implemented either with three moves or by swapping member by member.

The move constructor at lines 13 and 14 moves the data representation from `rhs` into `*this`; then it sets `rhs`' primitive data (including pointers) to a valid but easily destroyed state. Note that if there is non-primitive data, then that data must be moved in the initialization list. For example, if there were also `vector<string> items`, then the constructor would be:

```
IntCell( IntCell && rhs ) : storedValue{ rhs.storedValue },        // Move constructor
                            items{ std::move( rhs.items ) }
  { rhs.storedValue = nullptr; }
```