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

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

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.

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

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

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

$$N = P_1 P_2 P_3 \cdots P_k + 1$$

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]

```
1   void printOut( int n )  // Print nonnegative n
2   {
3       if( n >= 10 )
4           printOut( n / 10 );
5       printDigit( n % 10 );
6   }
```

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.

```
1  /**
2   * A class for simulating an integer memory cell.
3   */
4  class IntCell
5  {
6    public:
7      /**
8       * Construct the IntCell.
9       * Initial value is 0.
10      */
11     IntCell( )
12       { storedValue = 0; }
13
14     /**
15      * Construct the IntCell.
16      * Initial value is initialValue.
17      */
18     IntCell( int initialValue )
19       { storedValue = initialValue; }
20
21     /**
22      * Return the stored value.
23      */
24     int read( )
25       { return storedValue; }
26
27     /**
28      * Change the stored value to x.
29      */
30     void write( int x )
31       { storedValue = x; }
32
33   private:
34     int storedValue;
35 };
```

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

```
1  #ifndef IntCell_H
2  #define IntCell_H
3
4  /**
5   * A class for simulating an integer memory cell.
6   */
7  class IntCell
8  {
9    public:
10     explicit IntCell( int initialValue = 0 );
11     int read( ) const;
12     void write( int x );
13
14   private:
15     int storedValue;
16 };
17
18 #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.

```
1  #include "IntCell.h"
2
3  /**
4   * Construct the IntCell with initialValue
5   */
6  IntCell::IntCell( int initialValue ) : storedValue{ initialValue }
7  {
8  }
9
10 /**
11  * Return the stored value.
12  */
13 int IntCell::read( ) const
14 {
15     return storedValue;
16 }
17
18 /**
19  * Store x.
20  */
21 void IntCell::write( int x )
22 {
23     storedValue = x;
24 }
```

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

```
1  #include <iostream>
2  #include "IntCell.h"
3  using namespace std;
4
5  int main( )
6  {
7      IntCell m;
8
9      m.write( 5 );
10     cout << "Cell contents: " << m.read( ) << endl;
11
12     return 0;
13 }
```

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.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main( )
6  {
7      vector<int> squares( 100 );
8
9      for( int i = 0; i < squares.size( ); ++i )
10         squares[ i ] = i * i;
11
12     for( int i = 0; i < squares.size( ); ++i )
13         cout << i << " " << squares[ i ] << endl;
14
15     return 0;
16 }
```

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 `vectors`. In older C++, `vectors` 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.

```
1     vector<int> partialSum( const vector<int> & arr )
2     {
3         vector<int> result( arr.size( ) );
4
5         result[ 0 ] = arr[ 0 ];
6         for( int i = 1; i < arr.size( ); ++i )
7             result[ i ] = result[ i - 1 ] + arr[ i ];
8
9         return result;
10    }
11
12        vector<int> vec;
13          ...
14        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.

```
1   void swap( double & x, double & y )
2   {
3       double tmp = x;
4       x = y;
5       y = tmp;
6   }
7
8   void swap( vector<string> & x, vector<string> & y )
9   {
10      vector<string> tmp = x;
11      x = y;
12      y = tmp;
13  }
```

Figure 1.14 Swapping by three copies

```
1   void swap( vector<string> & x, vector<string> & y )
2   {
3       vector<string> tmp = static_cast<vector<string> &&>( x );
4       x = static_cast<vector<string> &&>( y );
5       y = static_cast<vector<string> &&>( tmp );
6   }
7
8   void swap( vector<string> & x, vector<string> & y )
9   {
10      vector<string> tmp = std::move( x );
11      x = std::move( y );
12      y = std::move( tmp );
13  }
```

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 `new`s, 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.

```
1    class IntCell
2    {
3      public:
4        explicit IntCell( int initialValue = 0 )
5          { storedValue = new int{ initialValue }; }
6
7        int read( ) const
8          { return *storedValue; }
9        void write( int x )
10         { *storedValue = x; }
11
12     private:
13       int *storedValue;
14   };
```

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

```
1     int f( )
2     {
3         IntCell a{ 2 };
4         IntCell b = a;
5         IntCell c;
6
7         c = b;
8         a.write( 4 );
9         cout << a.read( ) << endl << b.read( ) << endl << c.read( ) << endl;
10
11        return 0;
12    }
```

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; }
```

```
1   class IntCell
2   {
3     public:
4       explicit IntCell( int initialValue = 0 )
5         { storedValue = new int{ initialValue }; }
6
7       ~IntCell( )                                              // Destructor
8         { delete storedValue; }
9
10      IntCell( const IntCell & rhs )                           // Copy constructor
11        { storedValue = new int{ *rhs.storedValue }; }
12
13      IntCell( IntCell && rhs ) : storedValue{ rhs.storedValue }  // Move constructor
14        { rhs.storedValue = nullptr; }
15
16      IntCell & operator= ( const IntCell & rhs )              // Copy assignment
17      {
18          if( this != &rhs )
19              *storedValue = *rhs.storedValue;
20          return *this;
21      }
22
23      IntCell & operator= ( IntCell && rhs )                   // Move assignment
24      {
25          std::swap( storedValue, rhs.storedValue );
26          return *this;
27      }
28
29      int read( ) const
30        { return *storedValue; }
31      void write( int x )
32        { *storedValue = x; }
33
34    private:
35      int *storedValue;
36  };
```

Figure 1.18 Data member is a pointer; big-five is written

Finally, the move assignment operator at lines 23–27 is implemented as a member-by-member swap. Note that sometimes it is implemented as a single swap of objects in the same manner as the copy assignment operator, but that only works if swap itself is implemented as a member-by-member swap. If swap is implemented as three moves, then we would have mutual nonterminating recursion.

1.5.7 C-style Arrays and Strings

The C++ language provides a built-in C-style array type. To declare an array, `arr1`, of 10 integers, one writes:

```
int arr1[ 10 ];
```

arr1 is actually a pointer to memory that is large enough to store 10 ints, rather than a first-class array type. Applying = to arrays is thus an attempt to copy two pointer values rather than the entire array, and with the declaration above, it is illegal, because arr1 is a constant pointer. When arr1 is passed to a function, only the value of the pointer is passed; information about the size of the array is lost. Thus, the size must be passed as an additional parameter. There is no index range checking, since the size is unknown.

In the declaration above, the size of the array must be known at compile time. A variable cannot replace 10. If the size is unknown, we must explicitly declare a pointer and allocate memory via new[]. For instance,

```
int *arr2 = new int[ n ];
```

Now arr2 behaves like arr1, except that it is not a constant pointer. Thus, it can be made to point at a larger block of memory. However, because memory has been dynamically allocated, at some point it must be freed with delete[]:

```
delete [ ] arr2;
```

Otherwise, a memory leak will result, and the leak could be significant if the array is large.

Built-in C-style strings are implemented as an array of characters. To avoid having to pass the length of the string, the special null-terminator '\0' is used as a character that signals the logical end of the string. Strings are copied by strcpy, compared with strcmp, and their length can be determined by strlen. Individual characters can be accessed by the array indexing operator. These strings have all the problems associated with arrays, including difficult memory management, compounded by the fact that when strings are copied, it is assumed that the target array is large enough to hold the result. When it is not, difficult debugging ensues, often because room has not been left for the null terminator.

The standard vector class and string class are implemented by hiding the behavior of the built-in C-style array and string. Chapter 3 discusses the vector class implementation. It is almost always better to use the vector and string class, but you may be forced to use the C-style when interacting with library routines that are designed to work with both C and C++. It also is occasionally necessary (but this is rare) to use the C-style in a section of code that must be optimized for speed.

1.6 Templates

Consider the problem of finding the largest item in an array of items. A simple algorithm is the sequential scan, in which we examine each item in order, keeping track of the maximum. As is typical of many algorithms, the sequential scan algorithm is type independent. By type independent, we mean that the logic of this algorithm does not depend on the type of items that are stored in the array. The same logic works for an array of integers, floating-point numbers, or any type for which comparison can be meaningfully defined.

Throughout this text, we will describe algorithms and data structures that are type independent. When we write C++ code for a type-independent algorithm or data structure, we would prefer to write the code once rather than recode it for each different type.

In this section, we will describe how type-independent algorithms (also known as generic algorithms) are written in C++ using the **template**. We begin by discussing function templates. Then we examine class templates.

1.6.1 Function Templates

Function templates are generally very easy to write. A **function template** is not an actual function, but instead is a pattern for what could become a function. Figure 1.19 illustrates a function template `findMax`. The line containing the `template` declaration indicates that `Comparable` is the template argument: It can be replaced by any type to generate a function. For instance, if a call to `findMax` is made with a `vector<string>` as parameter, then a function will be generated by replacing `Comparable` with `string`.

Figure 1.20 illustrates that function templates are expanded automatically as needed. It should be noted that an expansion for each new type generates additional code; this is known as **code bloat** when it occurs in large projects. Note also that the call `findMax(v4)` will result in a compile-time error. This is because when `Comparable` is replaced by `IntCell`, line 12 in Figure 1.19 becomes illegal; there is no `<` function defined for `IntCell`. Thus, it is customary to include, prior to any template, comments that explain what assumptions are made about the template argument(s). This includes assumptions about what kinds of constructors are required.

Because template arguments can assume any class type, when deciding on parameter-passing and return-passing conventions, it should be assumed that template arguments are not primitive types. That is why we have returned by constant reference.

Not surprisingly, there are many arcane rules that deal with function templates. Most of the problems occur when the template cannot provide an exact match for the parameters but can come close (through implicit type conversions). There must be ways to resolve

```
1  /**
2   * Return the maximum item in array a.
3   * Assumes a.size( ) > 0.
4   * Comparable objects must provide operator< and operator=
5   */
6  template <typename Comparable>
7  const Comparable & findMax( const vector<Comparable> & a )
8  {
9      int maxIndex = 0;
10
11     for( int i = 1; i < a.size( ); ++i )
12         if( a[ maxIndex ] < a[ i ] )
13             maxIndex = i;
14     return a[ maxIndex ];
15 }
```

Figure 1.19 `findMax` function template

```
int main( )
{
    vector<int>     v1( 37 );
    vector<double>  v2( 40 );
    vector<string>  v3( 80 );
    vector<IntCell> v4( 75 );

    // Additional code to fill in the vectors not shown

    cout << findMax( v1 ) << endl;  // OK: Comparable = int
    cout << findMax( v2 ) << endl;  // OK: Comparable = double
    cout << findMax( v3 ) << endl;  // OK: Comparable = string
    cout << findMax( v4 ) << endl;  // Illegal; operator< undefined

    return 0;
}
```

Figure 1.20 Using `findMax` function template

ambiguities, and the rules are quite complex. Note that if there is a nontemplate and a template and both match, then the nontemplate gets priority. Also note that if there are two equally close approximate matches, then the code is illegal and the compiler will declare an ambiguity.

1.6.2 Class Templates

In the simplest version, a class template works much like a function template. Figure 1.21 shows the `MemoryCell` template. `MemoryCell` is like the `IntCell` class, but works for any type

```
/**
 * A class for simulating a memory cell.
 */
template <typename Object>
class MemoryCell
{
  public:
    explicit MemoryCell( const Object & initialValue = Object{ } )
      : storedValue{ initialValue } { }
    const Object & read( ) const
      { return storedValue; }
    void write( const Object & x )
      { storedValue = x; }
  private:
    Object storedValue;
};
```

Figure 1.21 `MemoryCell` class template without separation

```
1  int main( )
2  {
3      MemoryCell<int>    m1;
4      MemoryCell<string> m2{ "hello" };
5
6      m1.write( 37 );
7      m2.write( m2.read( ) + "world" );
8      cout << m1.read( ) << end1 << m2.read( ) << end1;
9
10     return 0;
11 }
```

Figure 1.22 Program that uses `MemoryCell` class template

`Object`, provided that `Object` has a zero-parameter constructor, a copy constructor, and a copy assignment operator.

Notice that `Object` is passed by constant reference. Also, notice that the default parameter for the constructor is not 0, because 0 might not be a valid `Object`. Instead, the default parameter is the result of constructing an `Object` with its zero-parameter constructor.

Figure 1.22 shows how the `MemoryCell` can be used to store objects of both primitive and class types. Notice that `MemoryCell` is not a class; it is only a class template. `MemoryCell<int>` and `MemoryCell<string>` are the actual classes.

If we implement class templates as a single unit, then there is very little syntax baggage. Many class templates are, in fact, implemented this way because, currently, separate compilation of templates does not work well on many platforms. Therefore, in many cases, the entire class, with its implementation, must be placed in a `.h` file. Popular implementations of the STL follow this strategy.

An alternative, discussed in Appendix A, is to separate the interface and implementation of the class templates. This adds extra syntax and baggage and historically has been difficult for compilers to handle cleanly. To avoid the extra syntax throughout this text, we provide, when necessary, in the online code, class templates with no separation of interface and implementation. In the figures, the interface is shown as if separate compilation was used, but the member function implementations are shown as if separate compilation was avoided. This allows us to avoid focusing on syntax.

1.6.3 `Object`, `Comparable`, and an Example

In this text, we repeatedly use `Object` and `Comparable` as generic types. `Object` is assumed to have a zero-parameter constructor, an `operator=`, and a copy constructor. `Comparable`, as suggested in the `findMax` example, has additional functionality in the form of `operator<` that can be used to provide a total order.[5]

[5] Some of the data structures in Chapter 12 use `operator==` in addition to `operator<`. Note that for the purpose of providing a total order, `a==b` if both `a<b` and `b<a` are `false`; thus the use of `operator==` is simply for convenience.

```
1   class Square
2   {
3     public:
4       explicit Square( double s = 0.0 ) : side{ s }
5         { }
6
7       double getSide( ) const
8         { return side; }
9       double getArea( ) const
10        { return side * side; }
11      double getPerimeter( ) const
12        { return 4 * side; }
13
14      void print( ostream & out = cout ) const
15        { out << "(square " << getSide( ) << ")"; }
16      bool operator< ( const Square & rhs ) const
17        { return getSide( ) < rhs.getSide( ); }
18
19    private:
20      double side;
21  };
22
23      // Define an output operator for Square
24  ostream & operator<< ( ostream & out, const Square & rhs )
25  {
26      rhs.print( out );
27      return out;
28  }
29
30  int main( )
31  {
32      vector<Square> v = { Square{ 3.0 }, Square{ 2.0 }, Square{ 2.5 } };
33
34      cout << "Largest square: " << findMax( v ) << endl;
35
36      return 0;
37  }
```

Figure 1.23 `Comparable` can be a class type, such as `Square`

Figure 1.23 shows an example of a class type that implements the functionality required of `Comparable` and illustrates **operator overloading.** Operator overloading allows us to define the meaning of a built-in operator. The `Square` class represents a square by storing the length of a side and defines `operator<`. The `Square` class also provides a zero-parameter constructor, `operator=`, and copy constructor (all by default). Thus, it has enough to be used as a `Comparable` in `findMax`.

Figure 1.23 shows a minimal implementation and also illustrates the widely used idiom for providing an output function for a new class type. The idiom is to provide a `public` member function, named `print`, that takes an `ostream` as a parameter. That `public` member function can then be called by a global, nonclass function, `operator<<`, that accepts an `ostream` and an object to output.

1.6.4 Function Objects

In Section 1.6.1, we showed how function templates can be used to write generic algorithms. As an example, the function template in Figure 1.19 can be used to find the maximum item in an array.

However, the template has an important limitation: It works only for objects that have an `operator<` function defined, and it uses that `operator<` as the basis for all comparison decisions. In many situations, this approach is not feasible. For instance, it is a stretch to presume that a `Rectangle` class will implement `operator<`, and even if it does, the `compareTo` method that it has might not be the one we want. For instance, given a 2-by-10 rectangle and a 5-by-5 rectangle, which is the larger rectangle? The answer would depend on whether we are using area or width to decide. Or perhaps if we are trying to fit the rectangle through an opening, the larger rectangle is the rectangle with the larger minimum dimension. As a second example, if we wanted to find the maximum string (alphabetically last) in an array of strings, the default `operator<` does not ignore case distinctions, so "ZEBRA" would be considered to precede "alligator" alphabetically, which is probably not what we want. A third example would occur if we had an array of pointers to objects (which would be common in advanced C++ programs that make use of a feature known as inheritance, which we do not make much use of in this text).

The solution, in these cases, is to rewrite `findMax` to accept as parameters an array of objects and a comparison function that explains how to decide which of two objects is the larger and which is the smaller. In effect, the array objects no longer know how to compare themselves; instead, this information is completely decoupled from the objects in the array.

An ingenious way to pass functions as parameters is to notice that an object contains both data and member functions, so we can define a class with no data and one member function, and pass an instance of the class. In effect, a function is being passed by placing it inside an object. This object is commonly known as a **function object.**

Figure 1.24 shows the simplest implementation of the function object idea. `findMax` takes a second parameter, which is a generic type. In order for the `findMax` template to expand without error, the generic type must have a member function named `isLessThan`, which takes two parameters of the first generic type (`Object`) and returns a `bool`. Otherwise, an error will be generated at line 9 when the template expansion is attempted by the compiler. At line 25, we can see that `findMax` is called by passing an array of string and an object that contains an `isLessThan` method with two strings as parameters.

C++ function objects are implemented using this basic idea, but with some fancy syntax. First, instead of using a function with a name, we use operator overloading. Instead of the function being `isLessThan`, it is `operator()`. Second, when invoking `operator()`,

```
1   // Generic findMax, with a function object, Version #1.
2   // Precondition: a.size( ) > 0.
3   template <typename Object, typename Comparator>
4   const Object & findMax( const vector<Object> & arr, Comparator cmp )
5   {
6       int maxIndex = 0;
7
8       for( int i = 1; i < arr.size( ); ++i )
9           if( cmp.isLessThan( arr[ maxIndex ], arr[ i ] ) )
10              maxIndex = i;
11
12      return arr[ maxIndex ];
13  }
14
15  class CaseInsensitiveCompare
16  {
17    public:
18      bool isLessThan( const string & lhs, const string & rhs ) const
19        { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
20  };
21
22  int main( )
23  {
24      vector<string> arr = { "ZEBRA", "alligator", "crocodile" };
25      cout << findMax( arr, CaseInsensitiveCompare{ } ) << endl;
26
27      return 0;
28  }
```

Figure 1.24 Simplest idea of using a function object as a second parameter to `findMax`; output is ZEBRA

`cmp.operator()(x,y)` can be shortened to `cmp(x,y)` (in other words, it looks like a function call, and consequently `operator()` is known as the **function call operator**). As a result, the name of the parameter can be changed to the more meaningful `isLessThan`, and the call is `isLessThan(x,y)`. Third, we can provide a version of `findMax` that works without a function object. The implementation uses the Standard Library function object template `less` (defined in header file *functional*) to generate a function object that imposes the normal default ordering. Figure 1.25 shows the implementation using the more typical, somewhat cryptic, C++ idioms.

In Chapter 4, we will give an example of a class that needs to order the items it stores. We will write most of the code using `Comparable` and show the adjustments needed to use the function objects. Elsewhere in the book, we will avoid the detail of function objects to keep the code as simple as possible, knowing that it is not difficult to add function objects later.

```
1   // Generic findMax, with a function object, C++ style.
2   // Precondition: a.size( ) > 0.
3   template <typename Object, typename Comparator>
4   const Object & findMax( const vector<Object> & arr, Comparator isLessThan )
5   {
6       int maxIndex = 0;
7
8       for( int i = 1; i < arr.size( ); ++i )
9           if( isLessThan( arr[ maxIndex ], arr[ i ] ) )
10              maxIndex = i;
11
12      return arr[ maxIndex ];
13  }
14
15  // Generic findMax, using default ordering.
16  #include <functional>
17  template <typename Object>
18  const Object & findMax( const vector<Object> & arr )
19  {
20      return findMax( arr, less<Object>{ } );
21  }
22
23  class CaseInsensitiveCompare
24  {
25    public:
26      bool operator( )( const string & lhs, const string & rhs ) const
27        { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
28  };
29
30  int main( )
31  {
32      vector<string> arr = { "ZEBRA", "alligator", "crocodile" };
33
34      cout << findMax( arr, CaseInsensitiveCompare{ } ) << endl;
35      cout << findMax( arr ) << endl;
36
37      return 0;
38  }
```

Figure 1.25 Using a function object C++ style, with a second version of `findMax`; output is ZEBRA, then crocodile

1.6.5 Separate Compilation of Class Templates

Like regular classes, class templates can be implemented either entirely in their declarations, or we can separate the interface from the implementation. However, compiler support for separate compilation of templates historically has been weak and platform-specific. Thus, in many cases, the entire class template with its implementation is placed in a single header file. Popular implementations of the Standard Library follow this strategy to implement class templates.

Appendix A describes the mechanics involved in the separate compilation of templates. The declaration of the interface for a template is exactly what you would expect: The member functions end with a single semicolon, instead of providing an implementation. But as shown in Appendix A, the implementation of the member functions can introduce complicated-looking syntax, especially for complicated functions like `operator=`. Worse, when compiling, the compiler will often complain about missing functions, and avoiding this problem requires platform-specific solutions.

Consequently, in the online code that accompanies the text, we implement all class templates entirely in its declaration in a single header file. We do so because it seems to be the only way to avoid compilation problems across platforms. In the text, when illustrating the code, we provide the class interface as if separate compilation was in order, since that is easily presentable, but implementations are shown as in the online code. In a platform-specific manner, one can mechanically transform our single header file implementations into separate compilation implementations if desired. See Appendix A for some of the different scenarios that might apply.

1.7 Using Matrices

Several algorithms in Chapter 10 use two-dimensional arrays, which are popularly known as matrices. The C++ library does not provide a `matrix` class. However, a reasonable `matrix` class can quickly be written. The basic idea is to use a vector of vectors. Doing this requires additional knowledge of operator overloading. For the `matrix`, we define `operator[]`, namely, the array-indexing operator. The `matrix` class is given in Figure 1.26.

1.7.1 The Data Members, Constructor, and Basic Accessors

The matrix is represented by an `array` data member that is declared to be a `vector` of `vector<Object>`. The constructor first constructs `array` as having `rows` entries each of type `vector<Object>` that is constructed with the zero-parameter constructor. Thus, we have `rows` zero-length vectors of `Object`.

The body of the constructor is then entered, and each row is resized to have `cols` columns. Thus the constructor terminates with what appears to be a two-dimensional array. The `numrows` and `numcols` accessors are then easily implemented, as shown.

```
1   #ifndef MATRIX_H
2   #define MATRIX_H
3
4   #include <vector>
5   using namespace std;
6
7   template <typename Object>
8   class matrix
9   {
10    public:
11      matrix( int rows, int cols ) : array( rows )
12      {
13          for( auto & thisRow : array )
14              thisRow.resize( cols );
15      }
16
17      matrix( vector<vector<Object>> v ) : array{ v }
18        { }
19      matrix( vector<vector<Object>> && v ) : array{ std::move( v ) }
20        { }
21
22      const vector<Object> & operator[]( int row ) const
23        { return array[ row ]; }
24      vector<Object> & operator[]( int row )
25        { return array[ row ]; }
26
27      int numrows( ) const
28        { return array.size( ); }
29      int numcols( ) const
30        { return numrows( ) ? array[ 0 ].size( ) : 0; }
31    private:
32      vector<vector<Object>> array;
33  };
34  #endif
```

Figure 1.26 A complete `matrix` class

1.7.2 operator[]

The idea of `operator[]` is that if we have a `matrix m`, then `m[i]` should return a vector corresponding to row `i` of `matrix m`. If this is done, then `m[i][j]` will give the entry in position `j` for vector `m[i]`, using the normal `vector` indexing operator. Thus, the `matrix operator[]` returns a `vector<Object>` rather than an `Object`.

We now know that `operator[]` should return an entity of type `vector<Object>`. Should we use return-by-value, return-by-reference, or return-by-constant-reference? Immediately we eliminate return-by-value, because the returned entity is large but guaranteed to exist after the call. Thus, we are down to return-by-reference or return-by-constant-reference. Consider the following method (ignore the possibility of aliasing or incompatible sizes, neither of which affects the algorithm):

```
void copy( const matrix<int> & from, matrix<int> & to )
{
    for( int i = 0; i < to.numrows( ); ++i )
        to[ i ] = from[ i ];
}
```

In the `copy` function, we attempt to copy each row in `matrix from` into the corresponding row in `matrix to`. Clearly, if `operator[]` returns a constant reference, then `to[i]` cannot appear on the left side of the assignment statement. Thus, it appears that `operator[]` should return a reference. However, if we did that, then an expression such as `from[i]=to[i]` would compile, since `from[i]` would not be a constant vector, even though `from` was a constant matrix. That cannot be allowed in a good design.

So what we really need is for `operator[]` to return a constant reference for `from`, but a plain reference for `to`. In other words, we need two versions of `operator[]`, which differ only in their return types. That is not allowed. However, there is a loophole: Since member function const-ness (i.e., whether a function is an accessor or a mutator) is part of the signature, we can have the accessor version of `operator[]` return a constant reference, and have the mutator version return the simple reference. Then, all is well. This is shown in Figure 1.26.

1.7.3 Big-Five

These are all taken care of automatically, because the `vector` has taken care of it. Therefore, this is all the code needed for a fully functioning `matrix` class.

Summary

This chapter sets the stage for the rest of the book. The time taken by an algorithm confronted with large amounts of input will be an important criterion for deciding if it is a good algorithm. (Of course, correctness is most important.) We will begin to address these issues in the next chapter and will use the mathematics discussed here to establish a formal model.

Exercises

1.1 Write a program to solve the selection problem. Let $k = N/2$. Draw a table showing the running time of your program for various values of N.

1.2 Write a program to solve the word puzzle problem.

1.3 Write a function to output an arbitrary `double` number (which might be negative) using only `printDigit` for I/O.

1.4 C++ allows statements of the form

```
#include filename
```

which reads *filename* and inserts its contents in place of the *include* statement. *Include* statements may be nested; in other words, the file *filename* may itself contain an *include* statement, but, obviously, a file can't include itself in any chain. Write a program that reads in a file and outputs the file as modified by the *include* statements.

1.5 Write a recursive function that returns the number of 1 in the binary representation of N. Use the fact that this is equal to the number of 1 in the representation of $N/2$, plus 1, if N is odd.

1.6 Write the routines with the following declarations:

```
void permute( const string & str );
void permute( const string & str, int low, int high );
```

The first routine is a driver that calls the second and prints all the permutations of the characters in `string str`. If `str` is `"abc"`, then the strings that are output are `abc`, `acb`, `bac`, `bca`, `cab`, and `cba`. Use recursion for the second routine.

1.7 Prove the following formulas:

a. $\log X < X$ for all $X > 0$

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

1.8 Evaluate the following sums:

a. $\sum_{i=0}^{\infty} \frac{1}{4^i}$

b. $\sum_{i=0}^{\infty} \frac{i}{4^i}$

* c. $\sum_{i=0}^{\infty} \frac{i^2}{4^i}$

** d. $\sum_{i=0}^{\infty} \frac{i^N}{4^i}$

1.9 Estimate

$$\sum_{i=\lfloor N/2 \rfloor}^{N} \frac{1}{i}$$

* **1.10** What is $2^{100} \pmod 5$?

1.11 Let F_i be the Fibonacci numbers as defined in Section 1.2. Prove the following:

a. $\sum_{i=1}^{N-2} F_i = F_N - 2$

b. $F_N < \phi^N$, with $\phi = (1 + \sqrt{5})/2$

** c. Give a precise closed-form expression for F_N.

1.12 Prove the following formulas:

a. $\sum_{i=1}^{N} (2i - 1) = N^2$

b. $\sum_{i=1}^{N} i^3 = \left(\sum_{i=1}^{N} i \right)^2$

1.13 Design a class template, `Collection`, that stores a collection of `Objects` (in an array), along with the current size of the collection. Provide `public` functions `isEmpty`, `makeEmpty`, `insert`, `remove`, and `contains`. `contains(x)` returns `true` if and only if an `Object` that is equal to `x` is present in the collection.

1.14 Design a class template, `OrderedCollection`, that stores a collection of `Comparables` (in an array), along with the current size of the collection. Provide `public` functions `isEmpty`, `makeEmpty`, `insert`, `remove`, `findMin`, and `findMax`. `findMin` and `findMax` return references to the smallest and largest, respectively, `Comparable` in the collection. Explain what can be done if these operations are performed on an empty collection.

1.15 Define a `Rectangle` class that provides `getLength` and `getWidth`. Using the `findMax` routines in Figure 1.25, write a `main` that creates an array of `Rectangle` and finds the largest `Rectangle` first on the basis of area and then on the basis of perimeter.

1.16 For the `matrix` class, add a `resize` member function and zero-parameter constructor.

References

There are many good textbooks covering the mathematics reviewed in this chapter. A small subset is [1], [2], [3], [9], [14], and [16]. Reference [9] is specifically geared toward the analysis of algorithms. It is the first volume of a three-volume series that will be cited throughout this text. More advanced material is covered in [6].

Throughout this book, we will assume a knowledge of C++. For the most part, [15] describes the final draft standard of C++11, and, being written by the original designer of C++, remains the most authoritative. Another standard reference is [10]. Advanced topics in C++ are discussed in [5]. The two-part series [11, 12] gives a great discussion of the many pitfalls in C++. The Standard Template Library, which we will investigate throughout this text, is described in [13]. The material in Sections 1.4–1.7 is meant to serve as an overview of the features that we will use in this text. We also assume familiarity with pointers and recursion (the recursion summary in this chapter is meant to be a quick review). We will attempt to provide hints on their use where appropriate throughout the textbook. Readers not familiar with these should consult [17] or any good intermediate programming textbook.

General programming style is discussed in several books. Some of the classics are [4], [7], and [8].

1. M. O. Albertson and J. P. Hutchinson, *Discrete Mathematics with Algorithms,* John Wiley & Sons, New York, 1988.
2. Z. Bavel, *Math Companion for Computer Science,* Reston Publishing Co., Reston, Va., 1982.
3. R. A. Brualdi, *Introductory Combinatorics,* 5th ed., Pearson, Boston, Mass, 2009.
4. E. W. Dijkstra, *A Discipline of Programming,* Prentice Hall, Englewood Cliffs, N.J., 1976.
5. B. Eckel, *Thinking in C++*, 2d ed., Prentice Hall, Englewood Cliffs, N.J., 2002.
6. R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics,* Addison-Wesley, Reading, Mass., 1989.
7. D. Gries, *The Science of Programming,* Springer-Verlag, New York, 1981.

8. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style,* 2d ed., McGraw-Hill, New York, 1978.
9. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1997.
10. S. B. Lippman, J. Lajoie, and B. E. Moo, *C++ Primer*, 5th ed., Pearson, Boston, Mass., 2013.
11. S. Meyers, *50 Specific Ways to Improve Your Programs and Designs,* 3d ed., Addison-Wesley, Boston, Mass., 2005.
12. S. Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs,* Addison-Wesley, Reading, Mass., 1996.
13. D. R. Musser, G. J. Durge, and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library,* 2d ed., Addison-Wesley, Reading, Mass., 2001.
14. F. S. Roberts and B. Tesman, *Applied Combinatorics,* 2d ed., Prentice Hall, Englewood Cliffs, N.J., 2003.
15. B. Stroustrop, *The C++ Programming Language,* 4th ed., Pearson, Boston, Mass., 2013.
16. A. Tucker, *Applied Combinatorics,* 6th ed., John Wiley & Sons, New York, 2012.
17. M. A. Weiss, *Algorithms, Data Structures, and Problem Solving with C++*, 2nd ed., Addison-Wesley, Reading, Mass., 2000.

This page intentionally left blank

CHAPTER 2

Algorithm Analysis

An **algorithm** is a clearly specified set of simple instructions to be followed to solve a problem. Once an algorithm is given for a problem and decided (somehow) to be correct, an important step is to determine how much in the way of resources, such as time or space, the algorithm will require. An algorithm that solves a problem but requires a year is hardly of any use. Likewise, an algorithm that requires thousands of gigabytes of main memory is not (currently) useful on most machines.

In this chapter, we shall discuss . . .

- How to estimate the time required for a program.
- How to reduce the running time of a program from days or years to fractions of a second.
- The results of careless use of recursion.
- Very efficient algorithms to raise a number to a power and to compute the greatest common divisor of two numbers.

2.1 Mathematical Background

The analysis required to estimate the resource use of an algorithm is generally a theoretical issue, and therefore a formal framework is required. We begin with some mathematical definitions.

Throughout this book, we will use the following four definitions:

Definition 2.1
$T(N) = O(f(N))$ if there are positive *constants* c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Definition 2.2
$T(N) = \Omega(g(N))$ if there are positive *constants* c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.

Definition 2.3
$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

Definition 2.4
$T(N) = o(p(N))$ if, for all positive constants c, there exists an n_0 such that $T(N) < cp(N)$ when $N > n_0$. Less formally, $T(N) = o(p(N))$ if $T(N) = O(p(N))$ and $T(N) \neq \Theta(p(N))$.

The idea of these definitions is to establish a relative order among functions. Given two functions, there are usually points where one function is smaller than the other. So it does not make sense to claim, for instance, $f(N) < g(N)$. Thus, we compare their **relative rates of growth**. When we apply this to the analysis of algorithms, we shall see why this is the important measure.

Although $1{,}000N$ is larger than N^2 for small values of N, N^2 grows at a faster rate, and thus N^2 will eventually be the larger function. The turning point is $N = 1{,}000$ in this case. The first definition says that eventually there is some point n_0 past which $c \cdot f(N)$ is always at least as large as $T(N)$, so that if constant factors are ignored, $f(N)$ is at least as big as $T(N)$. In our case, we have $T(N) = 1{,}000N$, $f(N) = N^2$, $n_0 = 1{,}000$, and $c = 1$. We could also use $n_0 = 10$ and $c = 100$. Thus, we can say that $1{,}000N = O(N^2)$ (order N-squared). This notation is known as **Big-Oh notation**. Frequently, instead of saying "order . . . ," one says "Big-Oh . . . ."

If we use the traditional inequality operators to compare growth rates, then the first definition says that the growth rate of $T(N)$ is less than or equal to ($\leq$) that of $f(N)$. The second definition, $T(N) = \Omega(g(N))$ (pronounced "omega"), says that the growth rate of $T(N)$ is greater than or equal to ($\geq$) that of $g(N)$. The third definition, $T(N) = \Theta(h(N))$ (pronounced "theta"), says that the growth rate of $T(N)$ equals ($=$) the growth rate of $h(N)$. The last definition, $T(N) = o(p(N))$ (pronounced "little-oh"), says that the growth rate of $T(N)$ is less than ($<$) the growth rate of $p(N)$. This is different from Big-Oh, because Big-Oh allows the possibility that the growth rates are the same.

To prove that some function $T(N) = O(f(N))$, we usually do not apply these definitions formally but instead use a repertoire of known results. In general, this means that a proof (or determination that the assumption is incorrect) is a very simple calculation and should not involve calculus, except in extraordinary circumstances (not likely to occur in an algorithm analysis).

When we say that $T(N) = O(f(N))$, we are guaranteeing that the function $T(N)$ grows at a rate no faster than $f(N)$; thus $f(N)$ is an **upper bound** on $T(N)$. Since this implies that $f(N) = \Omega(T(N))$, we say that $T(N)$ is a **lower bound** on $f(N)$.

As an example, N^3 grows faster than N^2, so we can say that $N^2 = O(N^3)$ or $N^3 = \Omega(N^2)$. $f(N) = N^2$ and $g(N) = 2N^2$ grow at the same rate, so both $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$ are true. When two functions grow at the same rate, then the decision of whether or not to signify this with $\Theta()$ can depend on the particular context. Intuitively, if $g(N) = 2N^2$, then $g(N) = O(N^4)$, $g(N) = O(N^3)$, and $g(N) = O(N^2)$ are all technically correct, but the last option is the best answer. Writing $g(N) = \Theta(N^2)$ says not only that $g(N) = O(N^2)$ but also that the result is as good (tight) as possible.

Here are the important things to know:

Rule 1

If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then

(a) $T_1(N) + T_2(N) = O(f(N) + g(N))$ (intuitively and less formally it is $O(\max(f(N), g(N)))$),

(b) $T_1(N) * T_2(N) = O(f(N) * g(N))$.

Rule 2

If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$.

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 2.1 Typical growth rates

Rule 3
$\log^k N = O(N)$ for any constant k. This tells us that logarithms grow very slowly.

This information is sufficient to arrange most of the common functions by growth rate (see Fig. 2.1).

Several points are in order. First, it is very bad style to include constants or low-order terms inside a Big-Oh. Do not say $T(N) = O(2N^2)$ or $T(N) = O(N^2 + N)$. In both cases, the correct form is $T(N) = O(N^2)$. This means that in any analysis that will require a Big-Oh answer, all sorts of shortcuts are possible. Lower-order terms can generally be ignored, and constants can be thrown away. Considerably less precision is required in these cases.

Second, we can always determine the relative growth rates of two functions $f(N)$ and $g(N)$ by computing $\lim_{N\to\infty} f(N)/g(N)$, using L'Hôpital's rule if necessary.[1] The limit can have four possible values:

- The limit is 0: This means that $f(N) = o(g(N))$.
- The limit is $c \neq 0$: This means that $f(N) = \Theta(g(N))$.
- The limit is ∞: This means that $g(N) = o(f(N))$.
- The limit does not exist: There is no relation (this will not happen in our context).

Using this method almost always amounts to overkill. Usually the relation between $f(N)$ and $g(N)$ can be derived by simple algebra. For instance, if $f(N) = N \log N$ and $g(N) = N^{1.5}$, then to decide which of $f(N)$ and $g(N)$ grows faster, one really needs to determine which of $\log N$ and $N^{0.5}$ grows faster. This is like determining which of $\log^2 N$ or N grows faster. This is a simple problem, because it is already known that N grows faster than any power of a log. Thus, $g(N)$ grows faster than $f(N)$.

One stylistic note: It is bad to say $f(N) \leq O(g(N))$, because the inequality is implied by the definition. It is wrong to write $f(N) \geq O(g(N))$, because it does not make sense.

[1] L'Hôpital's rule states that if $\lim_{N\to\infty} f(N) = \infty$ and $\lim_{N\to\infty} g(N) = \infty$, then $\lim_{N\to\infty} f(N)/g(N) = \lim_{N\to\infty} f'(N)/g'(N)$, where $f'(N)$ and $g'(N)$ are the derivatives of $f(N)$ and $g(N)$, respectively.

As an example of the typical kinds of analyses that are performed, consider the problem of downloading a file over the Internet. Suppose there is an initial 3-sec delay (to set up a connection), after which the download proceeds at 1.5M(bytes)/sec. Then it follows that if the file is N megabytes, the time to download is described by the formula $T(N) = N/1.5 + 3$. This is a **linear function**. Notice that the time to download a 1,500M file (1,003 sec) is approximately (but not exactly) twice the time to download a 750M file (503 sec). This is typical of a linear function. Notice, also, that if the speed of the connection doubles, both times decrease, but the 1,500M file still takes approximately twice the time to download as a 750M file. This is the typical characteristic of linear-time algorithms, and it is the reason we write $T(N) = O(N)$, ignoring constant factors. (Although using big-theta would be more precise, Big-Oh answers are typically given.)

Observe, too, that this behavior is not true of all algorithms. For the first selection algorithm described in Section 1.1, the running time is controlled by the time it takes to perform a sort. For a simple sorting algorithm, such as the suggested bubble sort, when the amount of input doubles, the running time increases by a factor of four for large amounts of input. This is because those algorithms are not linear. Instead, as we will see when we discuss sorting, trivial sorting algorithms are $O(N^2)$, or quadratic.

2.2 Model

In order to analyze algorithms in a formal framework, we need a model of computation. Our model is basically a normal computer in which instructions are executed sequentially. Our model has the standard repertoire of simple instructions, such as addition, multiplication, comparison, and assignment, but, unlike the case with real computers, it takes exactly one time unit to do anything (simple). To be reasonable, we will assume that, like a modern computer, our model has fixed-size (say, 32-bit) integers and no fancy operations, such as matrix inversion or sorting, which clearly cannot be done in one time unit. We also assume infinite memory.

This model clearly has some weaknesses. Obviously, in real life, not all operations take exactly the same time. In particular, in our model, one disk reads counts the same as an addition, even though the addition is typically several orders of magnitude faster. Also, by assuming infinite memory, we ignore the fact that the cost of a memory access can increase when slower memory is used due to larger memory requirements.

2.3 What to Analyze

The most important resource to analyze is generally the running time. Several factors affect the running time of a program. Some, such as the compiler and computer used, are obviously beyond the scope of any theoretical model, so, although they are important, we cannot deal with them here. The other main factors are the algorithm used and the input to the algorithm.

Typically, the size of the input is the main consideration. We define two functions, $T_{\text{avg}}(N)$ and $T_{\text{worst}}(N)$, as the average and worst-case running time, respectively, used by an algorithm on input of size N. Clearly, $T_{\text{avg}}(N) \leq T_{\text{worst}}(N)$. If there is more than one input, these functions may have more than one argument.

Occasionally, the best-case performance of an algorithm is analyzed. However, this is often of little interest, because it does not represent typical behavior. Average-case performance often reflects typical behavior, while worst-case performance represents a guarantee for performance on any possible input. Notice also that, although in this chapter we analyze C++ code, these bounds are really bounds for the algorithms rather than programs. Programs are an implementation of the algorithm in a particular programming language, and almost always the details of the programming language do not affect a Big-Oh answer. If a program is running much more slowly than the algorithm analysis suggests, there may be an implementation inefficiency. This can occur in C++ when arrays are inadvertently copied in their entirety, instead of passed with references. Another extremely subtle example of this is in the last two paragraphs of Section 12.6. Thus in future chapters, we will analyze the algorithms rather than the programs.

Generally, the quantity required is the worst-case time, unless otherwise specified. One reason for this is that it provides a bound for all input, including particularly bad input, which an average-case analysis does not provide. The other reason is that average-case bounds are usually much more difficult to compute. In some instances, the definition of "average" can affect the result. (For instance, what is average input for the following problem?)

As an example, in the next section, we shall consider the following problem:

Maximum Subsequence Sum Problem

Given (possibly negative) integers $A_1, A_2, \ldots, A_N$, find the maximum value of $\sum_{k=i}^{j} A_k$. (For convenience, the maximum subsequence sum is 0 if all the integers are negative.)

Example:

For input -2, 11, -4, 13, -5, -2, the answer is 20 (A_2 through A_4).

This problem is interesting mainly because there are so many algorithms to solve it, and the performance of these algorithms varies drastically. We will discuss four algorithms to solve this problem. The running time on some computers (the exact computer is unimportant) for these algorithms is given in Figure 2.2.

There are several important things worth noting in this table. For a small amount of input, the algorithms all run in the blink of an eye. So if only a small amount of input is

	Algorithm Time			
Input Size	1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

Figure 2.2 Running times of several algorithms for maximum subsequence sum (in seconds)

expected, it might be silly to expend a great deal of effort to design a clever algorithm. On the other hand, there is a large market these days for rewriting programs that were written five years ago based on a no-longer-valid assumption of small input size. These programs are now too slow because they used poor algorithms. For large amounts of input, algorithm 4 is clearly the best choice (although algorithm 3 is still usable).

Second, the times given do not include the time required to read the input. For algorithm 4, the time merely to read the input from a disk is likely to be an order of magnitude larger than the time required to solve the problem. This is typical of many efficient algorithms. Reading the data is generally the bottleneck; once the data are read, the problem can be solved quickly. For inefficient algorithms this is not true, and significant computer resources must be used. Thus, it is important that, whenever possible, algorithms be efficient enough not to be the bottleneck of a problem.

Notice that for algorithm 4, which is linear, as the problem size increases by a factor of 10, so does the running time. Algorithm 2, which is quadratic, does not display this behavior; a tenfold increase in input size yields roughly a hundredfold (10^2) increase in running time. And algorithm 1, which is cubic, yields a thousandfold (10^3) increase in running time. We would expect algorithm 1 to take nearly 9,000 seconds (or two and a half hours) to complete for $N = 100{,}000$. Similarly, we would expect algorithm 2 to take roughly 333 seconds to complete for $N = 1{,}000{,}000$. However, it is possible that algorithm 2 could take somewhat longer to complete due to the fact that $N = 1{,}000{,}000$ could also yield slower memory accesses than $N = 100{,}000$ on modern computers, depending on the size of the memory cache.

Figure 2.3 shows the growth rates of the running times of the four algorithms. Even though this graph encompasses only values of N ranging from 10 to 100, the relative

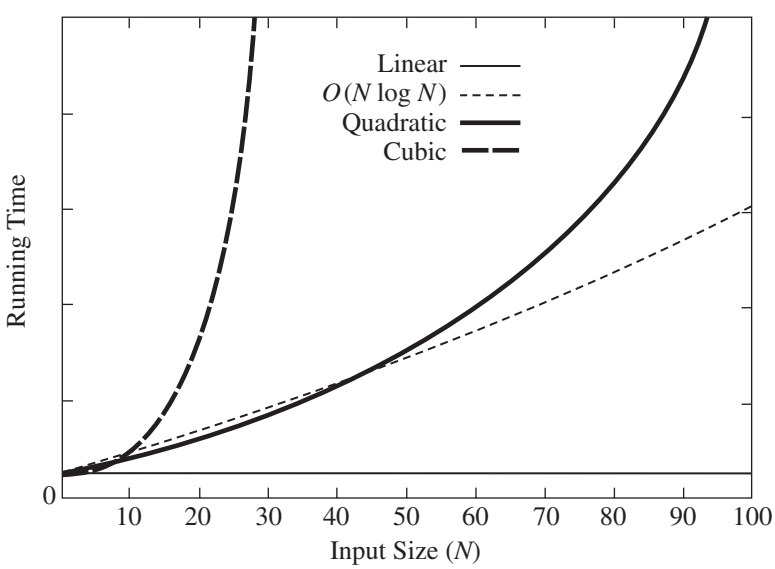

Figure 2.3 Plot (N vs. time) of various algorithms

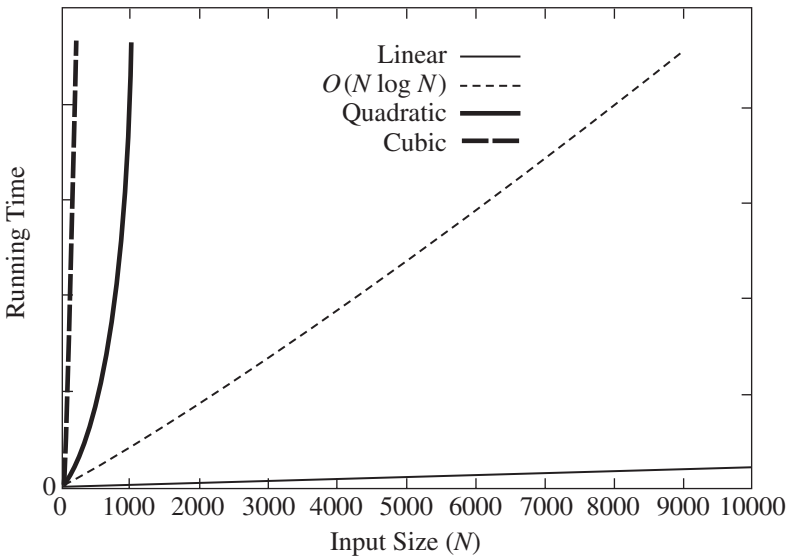

Figure 2.4 Plot (N vs. time) of various algorithms

growth rates are still evident. Although the graph for the $O(N \log N)$ seems linear, it is easy to verify that it is not by using a straightedge (or piece of paper). Although the graph for the $O(N)$ algorithm seems constant, this is only because for small values of N, the constant term is larger than the linear term. Figure 2.4 shows the performance for larger values. It dramatically illustrates how useless inefficient algorithms are for even moderately large amounts of input.

2.4 Running-Time Calculations

There are several ways to estimate the running time of a program. The previous table was obtained empirically. If two programs are expected to take similar times, probably the best way to decide which is faster is to code them both and run them!

Generally, there are several algorithmic ideas, and we would like to eliminate the bad ones early, so an analysis is usually required. Furthermore, the ability to do an analysis usually provides insight into designing efficient algorithms. The analysis also generally pinpoints the bottlenecks, which are worth coding carefully.

To simplify the analysis, we will adopt the convention that there are no particular units of time. Thus, we throw away leading constants. We will also throw away low-order terms, so what we are essentially doing is computing a Big-Oh running time. Since Big-Oh is an upper bound, we must be careful never to underestimate the running time of the program. In effect, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

2.4.1 A Simple Example

Here is a simple program fragment to calculate $\sum_{i=1}^{N} i^3$:

```
int sum( int n )
{
    int partialSum;

1       partialSum = 0;
2       for( int i = 1; i <= n; ++i )
3           partialSum += i * i * i;
4       return partialSum;
}
```

The analysis of this fragment is simple. The declarations count for no time. Lines 1 and 4 count for one unit each. Line 3 counts for four units per time executed (two multiplications, one addition, and one assignment) and is executed N times, for a total of $4N$ units. Line 2 has the hidden costs of initializing i, testing $i \leq N$, and incrementing i. The total cost of all these is 1 to initialize, $N + 1$ for all the tests, and N for all the increments, which is $2N + 2$. We ignore the costs of calling the function and returning, for a total of $6N + 4$. Thus, we say that this function is $O(N)$.

If we had to perform all this work every time we needed to analyze a program, the task would quickly become infeasible. Fortunately, since we are giving the answer in terms of Big-Oh, there are lots of shortcuts that can be taken without affecting the final answer. For instance, line 3 is obviously an $O(1)$ statement (per execution), so it is silly to count precisely whether it is two, three, or four units; it does not matter. Line 1 is obviously insignificant compared with the `for` loop, so it is silly to waste time here. This leads to several general rules.

2.4.2 General Rules

Rule 1—FOR loops

The running time of a `for` loop is at most the running time of the statements inside the `for` loop (including tests) times the number of iterations.

Rule 2—Nested loops

Analyze these inside out. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

As an example, the following program fragment is $O(N^2)$:

```
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        ++k;
```

Rule 3—Consecutive Statements

These just add (which means that the maximum is the one that counts; see rule 1 on page 52).

As an example, the following program fragment, which has $O(N)$ work followed by $O(N^2)$ work, is also $O(N^2)$:

```
for( i = 0; i < n; ++i )
    a[ i ] = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        a[ i ] += a[ j ] + i + j;
```

Rule 4—If/Else

For the fragment

```
if( condition )
    S1
else
    S2
```

the running time of an `if/else` statement is never more than the running time of the test plus the larger of the running times of S1 and S2.

Clearly, this can be an overestimate in some cases, but it is never an underestimate.

Other rules are obvious, but a basic strategy of analyzing from the inside (or deepest part) out works. If there are function calls, these must be analyzed first. If there are recursive functions, there are several options. If the recursion is really just a thinly veiled `for` loop, the analysis is usually trivial. For instance, the following function is really just a simple loop and is $O(N)$:

```
long factorial( int n )
{
    if( n <= 1 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

This example is really a poor use of recursion. When recursion is properly used, it is difficult to convert the recursion into a simple loop structure. In this case, the analysis will involve a recurrence relation that needs to be solved. To see what might happen, consider the following program, which turns out to be a terrible use of recursion:

```
    long fib( int n )
    {
1       if( n <= 1 )
2           return 1;
        else
3           return fib( n - 1 ) + fib( n - 2 );
    }
```

At first glance, this seems like a very clever use of recursion. However, if the program is coded up and run for values of N around 40, it becomes apparent that this program

is terribly inefficient. The analysis is fairly simple. Let $T(N)$ be the running time for the function call `fib(n)`. If $N = 0$ or $N = 1$, then the running time is some constant value, which is the time to do the test at line 1 and return. We can say that $T(0) = T(1) = 1$ because constants do not matter. The running time for other values of N is then measured relative to the running time of the base case. For $N > 2$, the time to execute the function is the constant work at line 1 plus the work at line 3. Line 3 consists of an addition and two function calls. Since the function calls are not simple operations, they must be analyzed by themselves. The first function call is `fib(n-1)` and hence, by the definition of T, requires $T(N - 1)$ units of time. A similar argument shows that the second function call requires $T(N - 2)$ units of time. The total time required is then $T(N - 1) + T(N - 2) + 2$, where the 2 accounts for the work at line 1 plus the addition at line 3. Thus, for $N \geq 2$, we have the following formula for the running time of `fib(n)`:

$$T(N) = T(N - 1) + T(N - 2) + 2$$

Since `fib(n)` $=$ `fib(n-1)` $+$ `fib(n-2)`, it is easy to show by induction that $T(N) \geq$ `fib(n)`. In Section 1.2.5, we showed that $fib(N) < (5/3)^N$. A similar calculation shows that (for $N > 4$) $fib(N) \geq (3/2)^N$, and so the running time of this program grows *exponentially*. This is about as bad as possible. By keeping a simple array and using a `for` loop, the running time can be reduced substantially.

This program is slow because there is a huge amount of redundant work being performed, violating the fourth major rule of recursion (the compound interest rule), which was presented in Section 1.3. Notice that the first call on line 3, `fib(n-1)`, actually computes `fib(n-2)` at some point. This information is thrown away and recomputed by the second call on line 3. The amount of information thrown away compounds recursively and results in the huge running time. This is perhaps the finest example of the maxim "Don't compute anything more than once" and should not scare you away from using recursion. Throughout this book, we shall see outstanding uses of recursion.

2.4.3 Solutions for the Maximum Subsequence Sum Problem

We will now present four algorithms to solve the maximum subsequence sum problem posed earlier. The first algorithm, which merely exhaustively tries all possibilities, is depicted in Figure 2.5. The indices in the `for` loop reflect the fact that in C++, arrays begin at 0 instead of 1. Also, the algorithm does not compute the actual subsequences; additional code is required to do this.

Convince yourself that this algorithm works (this should not take much convincing). The running time is $O(N^3)$ and is entirely due to lines 13 and 14, which consist of an $O(1)$ statement buried inside three nested `for` loops. The loop at line 8 is of size N.

The second loop has size $N - i$, which could be small but could also be of size N. We must assume the worst, with the knowledge that this could make the final bound a bit high. The third loop has size $j - i + 1$, which again we must assume is of size N. The total is $O(1 \cdot N \cdot N \cdot N) = O(N^3)$. Line 6 takes only $O(1)$ total, and lines 16 and 17 take only $O(N^2)$ total, since they are easy expressions inside only two loops.

```
1  /**
2   * Cubic maximum contiguous subsequence sum algorithm.
3   */
4  int maxSubSum1( const vector<int> & a )
5  {
6      int maxSum = 0;
7
8      for( int i = 0; i < a.size( ); ++i )
9          for( int j = i; j < a.size( ); ++j )
10         {
11             int thisSum = 0;
12
13             for( int k = i; k <= j; ++k )
14                 thisSum += a[ k ];
15
16             if( thisSum > maxSum )
17                 maxSum = thisSum;
18         }
19
20     return maxSum;
21 }
```

Figure 2.5 Algorithm 1

It turns out that a more precise analysis, taking into account the actual size of these loops, shows that the answer is $\Theta(N^3)$ and that our estimate above was a factor of 6 too high (which is all right, because constants do not matter). This is generally true in these kinds of problems. The precise analysis is obtained from the sum $\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}\sum_{k=i}^{j}1$, which tells how many times line 14 is executed. The sum can be evaluated inside out, using formulas from Section 1.2.3. In particular, we will use the formulas for the sum of the first N integers and first N squares. First we have

$$\sum_{k=i}^{j} 1 = j - i + 1$$

Next we evaluate

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

This sum is computed by observing that it is just the sum of the first $N - i$ integers. To complete the calculation, we evaluate

$$\sum_{i=0}^{N-1} \frac{(N-i+1)(N-i)}{2} = \sum_{i=1}^{N} \frac{(N-i+1)(N-i+2)}{2}$$
$$= \frac{1}{2}\sum_{i=1}^{N} i^2 - \left(N+\frac{3}{2}\right)\sum_{i=1}^{N} i + \frac{1}{2}(N^2+3N+2)\sum_{i=1}^{N} 1$$
$$= \frac{1}{2}\frac{N(N+1)(2N+1)}{6} - \left(N+\frac{3}{2}\right)\frac{N(N+1)}{2} + \frac{N^2+3N+2}{2}N$$
$$= \frac{N^3+3N^2+2N}{6}$$

We can avoid the cubic running time by removing a `for` loop. This is not always possible, but in this case there are an awful lot of unnecessary computations present in the algorithm. The inefficiency that the improved algorithm corrects can be seen by noticing that $\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$, so the computation at lines 13 and 14 in algorithm 1 is unduly expensive. Figure 2.6 shows an improved algorithm. Algorithm 2 is clearly $O(N^2)$; the analysis is even simpler than before.

There is a recursive and relatively complicated $O(N \log N)$ solution to this problem, which we now describe. If there didn't happen to be an $O(N)$ (linear) solution, this would be an excellent example of the power of recursion. The algorithm uses a "divide-and-conquer" strategy. The idea is to split the problem into two roughly equal subproblems,

```
1  /**
2   * Quadratic maximum contiguous subsequence sum algorithm.
3   */
4  int maxSubSum2( const vector<int> & a )
5  {
6      int maxSum = 0;
7
8      for( int i = 0; i < a.size( ); ++i )
9      {
10         int thisSum = 0;
11         for( int j = i; j < a.size( ); ++j )
12         {
13             thisSum += a[ j ];
14
15             if( thisSum > maxSum )
16                 maxSum = thisSum;
17         }
18     }
19
20     return maxSum;
21 }
```

Figure 2.6 Algorithm 2

which are then solved recursively. This is the "divide" part. The "conquer" stage consists of patching together the two solutions of the subproblems, and possibly doing a small amount of additional work, to arrive at a solution for the whole problem.

In our case, the maximum subsequence sum can be in one of three places. Either it occurs entirely in the left half of the input, or entirely in the right half, or it crosses the middle and is in both halves. The first two cases can be solved recursively. The last case can be obtained by finding the largest sum in the first half that includes the last element in the first half, and the largest sum in the second half that includes the first element in the second half. These two sums can then be added together. As an example, consider the following input:

First Half				Second Half			
4	−3	5	−2	−1	2	6	−2

The maximum subsequence sum for the first half is 6 (elements A_1 through A_3) and for the second half is 8 (elements A_6 through A_7).

The maximum sum in the first half that includes the last element in the first half is 4 (elements A_1 through A_4), and the maximum sum in the second half that includes the first element in the second half is 7 (elements A_5 through A_7). Thus, the maximum sum that spans both halves and goes through the middle is $4 + 7 = 11$ (elements A_1 through A_7).

We see, then, that among the three ways to form a large maximum subsequence, for our example, the best way is to include elements from both halves. Thus, the answer is 11. Figure 2.7 shows an implementation of this strategy.

The code for algorithm 3 deserves some comment. The general form of the call for the recursive function is to pass the input array along with the left and right borders, which delimits the portion of the array that is operated upon. A one-line driver program sets this up by passing the borders 0 and $N - 1$ along with the array.

Lines 8 to 12 handle the base case. If `left == right`, there is one element, and it is the maximum subsequence if the element is nonnegative. The case `left > right` is not possible unless N is negative (although minor perturbations in the code could mess this up). Lines 15 and 16 perform the two recursive calls. We can see that the recursive calls are always on a smaller problem than the original, although minor perturbations in the code could destroy this property. Lines 18 to 24 and 26 to 32 calculate the two maximum sums that touch the center divider. The sum of these two values is the maximum sum that spans both halves. The routine `max3` (not shown) returns the largest of the three possibilities.

Algorithm 3 clearly requires more effort to code than either of the two previous algorithms. However, shorter code does not always mean better code. As we have seen in the earlier table showing the running times of the algorithms, this algorithm is considerably faster than the other two for all but the smallest of input sizes.

The running time is analyzed in much the same way as for the program that computes the Fibonacci numbers. Let $T(N)$ be the time it takes to solve a maximum subsequence sum problem of size N. If $N = 1$, then the program takes some constant amount of time to execute lines 8 to 12, which we shall call one unit. Thus, $T(1) = 1$. Otherwise, the

```
1  /**
2   * Recursive maximum contiguous subsequence sum algorithm.
3   * Finds maximum sum in subarray spanning a[left..right].
4   * Does not attempt to maintain actual best sequence.
5   */
6  int maxSumRec( const vector<int> & a, int left, int right )
7  {
8      if( left == right )  // Base case
9          if( a[ left ] > 0 )
10             return a[ left ];
11         else
12             return 0;
13
14     int center = ( left + right ) / 2;
15     int maxLeftSum  = maxSumRec( a, left, center );
16     int maxRightSum = maxSumRec( a, center + 1, right );
17
18     int maxLeftBorderSum = 0, leftBorderSum = 0;
19     for( int i = center; i >= left; --i )
20     {
21         leftBorderSum += a[ i ];
22         if( leftBorderSum > maxLeftBorderSum )
23             maxLeftBorderSum = leftBorderSum;
24     }
25
26     int maxRightBorderSum = 0, rightBorderSum = 0;
27     for( int j = center + 1; j <= right; ++j )
28     {
29         rightBorderSum += a[ j ];
30         if( rightBorderSum > maxRightBorderSum )
31             maxRightBorderSum = rightBorderSum;
32     }
33
34     return max3( maxLeftSum, maxRightSum,
35                  maxLeftBorderSum + maxRightBorderSum );
36 }
37
38 /**
39  * Driver for divide-and-conquer maximum contiguous
40  * subsequence sum algorithm.
41  */
42 int maxSubSum3( const vector<int> & a )
43 {
44     return maxSumRec( a, 0, a.size( ) - 1 );
45 }
```

Figure 2.7 Algorithm 3

program must perform two recursive calls, the two `for` loops between lines 19 and 32, and some small amount of bookkeeping, such as lines 14 and 34. The two `for` loops combine to touch every element in the subarray, and there is constant work inside the loops, so the time expended in lines 19 to 32 is $O(N)$. The code in lines 8 to 14, 18, 26, and 34 is all a constant amount of work and can thus be ignored compared with $O(N)$. The remainder of the work is performed in lines 15 and 16. These lines solve two subsequence problems of size $N/2$ (assuming N is even). Thus, these lines take $T(N/2)$ units of time each, for a total of $2T(N/2)$. The total time for the algorithm then is $2T(N/2) + O(N)$. This gives the equations

$$T(1) = 1$$
$$T(N) = 2T(N/2) + O(N)$$

To simplify the calculations, we can replace the $O(N)$ term in the equation above with N; since $T(N)$ will be expressed in Big-Oh notation anyway, this will not affect the answer. In Chapter 7, we shall see how to solve this equation rigorously. For now, if $T(N) = 2T(N/2)+N$, and $T(1) = 1$, then $T(2) = 4 = 2*2$, $T(4) = 12 = 4*3$, $T(8) = 32 = 8*4$, and $T(16) = 80 = 16*5$. The pattern that is evident, and can be derived, is that if $N = 2^k$, then $T(N) = N*(k+1) = N\log N + N = O(N\log N)$.

This analysis assumes N is even, since otherwise $N/2$ is not defined. By the recursive nature of the analysis, it is really valid only when N is a power of 2, since otherwise we eventually get a subproblem that is not an even size, and the equation is invalid. When N is not a power of 2, a somewhat more complicated analysis is required, but the Big-Oh result remains unchanged.

In future chapters, we will see several clever applications of recursion. Here, we present a fourth algorithm to find the maximum subsequence sum. This algorithm is simpler to implement than the recursive algorithm and also is more efficient. It is shown in Figure 2.8.

It should be clear why the time bound is correct, but it takes a little thought to see why the algorithm actually works. To sketch the logic, note that like algorithms 1 and 2, `j` is representing the end of the current sequence, while `i` is representing the start of the current sequence. It happens that the use of `i` can be optimized out of the program if we do not need to know where the actual best subsequence is, but in designing the algorithm, let's pretend that `i` is needed and that we are trying to improve algorithm 2. One observation is that if `a[i]` is negative, then it cannot possibly be the start of the optimal subsequence, since any subsequence that begins by including `a[i]` would be improved by beginning with `a[i+1]`. Similarly, any negative subsequence cannot possibly be a prefix of the optimal subsequence (same logic). If, in the inner loop, we detect that the subsequence from `a[i]` to `a[j]` is negative, then we can advance `i`. The crucial observation is that not only can we advance `i` to `i+1`, but we can also actually advance it all the way to `j+1`. To see this, let `p` be any index between `i+1` and `j`. Any subsequence that starts at index `p` is not larger than the corresponding subsequence that starts at index `i` and includes the subsequence from `a[i]` to `a[p-1]`, since the latter subsequence is not negative (`j` is the first index that causes the subsequence starting at index `i` to become negative). Thus, advancing `i` to `j+1` is risk free; we cannot miss an optimal solution.

This algorithm is typical of many clever algorithms: The running time is obvious, but the correctness is not. For these algorithms, formal correctness proofs (more formal

```
/**
 * Linear-time maximum contiguous subsequence sum algorithm.
 */
int maxSubSum4( const vector<int> & a )
{
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.size( ); ++j )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
            maxSum = thisSum;
        else if( thisSum < 0 )
            thisSum = 0;
    }

    return maxSum;
}
```

Figure 2.8 Algorithm 4

than the sketch above) are almost always required; even then, many people still are not convinced. Also, many of these algorithms require trickier programming, leading to longer development. But when these algorithms work, they run quickly, and we can test much of the code logic by comparing it with an inefficient (but easily implemented) brute-force algorithm using small input sizes.

An extra advantage of this algorithm is that it makes only one pass through the data, and once `a[i]` is read and processed, it does not need to be remembered. Thus, if the array is on a disk or is being transmitted over the Internet, it can be read sequentially, and there is no need to store any part of it in main memory. Furthermore, at any point in time, the algorithm can correctly give an answer to the subsequence problem for the data it has already read (the other algorithms do not share this property). Algorithms that can do this are called **online algorithms**. An online algorithm that requires only constant space and runs in linear time is just about as good as possible.

2.4.4 Logarithms in the Running Time

The most confusing aspect of analyzing algorithms probably centers around the logarithm. We have already seen that some divide-and-conquer algorithms will run in $O(N \log N)$ time. Besides divide-and-conquer algorithms, the most frequent appearance of logarithms centers around the following general rule: *An algorithm is* $O(\log N)$ *if it takes constant* ($O(1)$) *time to cut the problem size by a fraction (which is usually* $\frac{1}{2}$*).* On the other hand, if constant time is required to merely reduce the problem by a constant *amount* (such as to make the problem smaller by 1), then the algorithm is $O(N)$.

It should be obvious that only special kinds of problems can be $O(\log N)$. For instance, if the input is a list of N numbers, an algorithm must take $\Omega(N)$ merely to read the input in. Thus, when we talk about $O(\log N)$ algorithms for these kinds of problems, we usually presume that the input is preread. We provide three examples of logarithmic behavior.

Binary Search

The first example is usually referred to as binary search.

Binary Search

Given an integer X and integers $A_0, A_1, \ldots, A_{N-1}$, which are presorted and already in memory, find i such that $A_i = X$, or return $i = -1$ if X is not in the input.

The obvious solution consists of scanning through the list from left to right and runs in linear time. However, this algorithm does not take advantage of the fact that the list is sorted and is thus not likely to be best. A better strategy is to check if X is the middle element. If so, the answer is at hand. If X is smaller than the middle element, we can apply the same strategy to the sorted subarray to the left of the middle element; likewise, if X is larger than the middle element, we look to the right half. (There is also the case of when to stop.) Figure 2.9 shows the code for binary search (the answer is `mid`). As usual, the code reflects C++'s convention that arrays begin with index 0.

```
1  /**
2   * Performs the standard binary search using two comparisons per level.
3   * Returns index where item is found or -1 if not found.
4   */
5  template <typename Comparable>
6  int binarySearch( const vector<Comparable> & a, const Comparable & x )
7  {
8      int low = 0, high = a.size( ) - 1;
9
10     while( low <= high )
11     {
12         int mid = ( low + high ) / 2;
13
14         if( a[ mid ] < x )
15             low = mid + 1;
16         else if( a[ mid ] > x )
17             high = mid - 1;
18         else
19             return mid;   // Found
20     }
21     return NOT_FOUND;     // NOT_FOUND is defined as -1
22 }
```

Figure 2.9 Binary search

Clearly, all the work done inside the loop takes $O(1)$ per iteration, so the analysis requires determining the number of times around the loop. The loop starts with `high - low` $= N - 1$ and finishes with `high - low` ≥ -1. Every time through the loop, the value `high - low` must be at least halved from its previous value; thus, the number of times around the loop is at most $\lceil \log(N-1) \rceil + 2$. (As an example, if `high - low` $= 128$, then the maximum values of `high - low` after each iteration are 64, 32, 16, 8, 4, 2, 1, 0, -1.) Thus, the running time is $O(\log N)$. Equivalently, we could write a recursive formula for the running time, but this kind of brute-force approach is usually unnecessary when you understand what is really going on and why.

Binary search can be viewed as our first data-structure implementation. It supports the `contains` operation in $O(\log N)$ time, but all other operations (in particular, `insert`) require $O(N)$ time. In applications where the data are static (i.e., insertions and deletions are not allowed), this could be very useful. The input would then need to be sorted once, but afterward accesses would be fast. An example is a program that needs to maintain information about the periodic table of elements (which arises in chemistry and physics). This table is relatively stable, as new elements are added infrequently. The element names could be kept sorted. Since there are only about 118 elements, at most eight accesses would be required to find an element. Performing a sequential search would require many more accesses.

Euclid's Algorithm

A second example is Euclid's algorithm for computing the greatest common divisor. The greatest common divisor (*gcd*) of two integers is the largest integer that divides both. Thus, $gcd(50, 15) = 5$. The algorithm in Figure 2.10 computes $gcd(M, N)$, assuming $M \geq N$. (If $N > M$, the first iteration of the loop swaps them.)

The algorithm works by continually computing remainders until 0 is reached. The last nonzero remainder is the answer. Thus, if $M = 1{,}989$ and $N = 1{,}590$, then the sequence of remainders is 399, 393, 6, 3, 0. Therefore, $gcd(1989, 1590) = 3$. As the example shows, this is a fast algorithm.

As before, estimating the entire running time of the algorithm depends on determining how long the sequence of remainders is. Although $\log N$ seems like a good answer, it is not at all obvious that the value of the remainder has to decrease by a constant factor,

```
1   long long gcd( long long m, long long n )
2   {
3       while( n != 0 )
4       {
5           long long rem = m % n;
6           m = n;
7           n = rem;
8       }
9       return m;
10  }
```

Figure 2.10 Euclid's algorithm

since we see that the remainder went from 399 to only 393 in the example. Indeed, the remainder *does not* decrease by a constant factor in one iteration. However, we can prove that after two iterations, the remainder is at most half of its original value. This would show that the number of iterations is at most $2 \log N = O(\log N)$ and establish the running time. This proof is easy, so we include it here. It follows directly from the following theorem.

Theorem 2.1
If $M > N$, then $M \bmod N < M/2$.

Proof
There are two cases. If $N \leq M/2$, then since the remainder is smaller than N, the theorem is true for this case. The other case is $N > M/2$. But then N goes into M once with a remainder $M - N < M/2$, proving the theorem.

One might wonder if this is the best bound possible, since $2 \log N$ is about 20 for our example, and only seven operations were performed. It turns out that the constant can be improved slightly, to roughly $1.44 \log N$, in the worst case (which is achievable if M and N are consecutive Fibonacci numbers). The average-case performance of Euclid's algorithm requires pages and pages of highly sophisticated mathematical analysis, and it turns out that the average number of iterations is about $(12 \ln 2 \ln N)/\pi^2 + 1.47$.

Exponentiation

Our last example in this section deals with raising an integer to a power (which is also an integer). Numbers that result from exponentiation are generally quite large, so an analysis works only if we can assume that we have a machine that can store such large integers (or a compiler that can simulate this). We will count the number of multiplications as the measurement of running time.

The obvious algorithm to compute X^N uses $N-1$ multiplications. A recursive algorithm can do better. $N \leq 1$ is the base case of the recursion. Otherwise, if N is even, we have $X^N = X^{N/2} \cdot X^{N/2}$, and if N is odd, $X^N = X^{(N-1)/2} \cdot X^{(N-1)/2} \cdot X$.

For instance, to compute X^{62}, the algorithm does the following calculations, which involve only nine multiplications:

$$X^3 = (X^2)X, X^7 = (X^3)^2X, X^{15} = (X^7)^2X, X^{31} = (X^{15})^2X, X^{62} = (X^{31})^2$$

The number of multiplications required is clearly at most $2 \log N$, because at most two multiplications (if N is odd) are required to halve the problem. Again, a recurrence formula can be written and solved. Simple intuition obviates the need for a brute-force approach.

Figure 2.11 implements this idea. It is sometimes interesting to see how much the code can be tweaked without affecting correctness. In Figure 2.11, lines 5 to 6 are actually unnecessary, because if N is 1, then line 10 does the right thing. Line 10 can also be rewritten as

```
10          return pow( x, n - 1 ) * x;
```

```
1   long long pow( long-long x, int n )
2   {
3       if( n == 0 )
4           return 1;
5       if( n == 1 )
6           return x;
7       if( isEven( n ) )
8           return pow( x * x, n / 2 );
9       else
10          return pow( x * x, n / 2 ) * x;
11  }
```

Figure 2.11 Efficient exponentiation

without affecting the correctness of the program. Indeed, the program will still run in $O(\log N)$, because the sequence of multiplications is the same as before. However, all of the following alternatives for line 8 are bad, even though they look correct:

```
8a          return pow( pow( x, 2 ), n / 2 );
8b          return pow( pow( x, n / 2 ), 2 );
8c          return pow( x, n / 2 ) * pow( x, n / 2 );
```

Both lines 8a and 8b are incorrect because when N is 2, one of the recursive calls to `pow` has 2 as the second argument. Thus no progress is made, and an infinite loop results (in an eventual crash).

Using line 8c affects the efficiency, because there are now two recursive calls of size $N/2$ instead of only one. An analysis will show that the running time is no longer $O(\log N)$. We leave it as an exercise to the reader to determine the new running time.

2.4.5 Limitations of Worst-Case Analysis

Sometimes the analysis is shown empirically to be an overestimate. If this is the case, then either the analysis needs to be tightened (usually by a clever observation), or it may be that the *average* running time is significantly less than the worst-case running time and no improvement in the bound is possible. For many complicated algorithms the worst-case bound is achievable by some bad input but is usually an overestimate in practice. Unfortunately, for most of these problems, an average-case analysis is extremely complex (in many cases still unsolved), and a worst-case bound, even though overly pessimistic, is the best analytical result known.

Summary

This chapter gives some hints on how to analyze the complexity of programs. Unfortunately, it is not a complete guide. Simple programs usually have simple analyses, but this is not always the case. As an example, later in the text we shall see a sorting algorithm (Shellsort, Chapter 7) and an algorithm for maintaining disjoint sets (Chapter 8), each of

which requires about 20 lines of code. The analysis of Shellsort is still not complete, and the disjoint set algorithm has an analysis that until recently was extremely difficult and require pages and pages of intricate calculations. Most of the analyses that we will encounter here will be simple and involve counting through loops.

An interesting kind of analysis, which we have not touched upon, is lower-bound analysis. We will see an example of this in Chapter 7, where it is proved that any algorithm that sorts by using only comparisons requires $\Omega(N \log N)$ comparisons in the worst case. Lower-bound proofs are generally the most difficult, because they apply not to an algorithm but to a class of algorithms that solve a problem.

We close by mentioning that some of the algorithms described here have real-life application. The *gcd* algorithm and the exponentiation algorithm are both used in cryptography. Specifically, a 600-digit number is raised to a large power (usually another 600-digit number), with only the low 600 or so digits retained after each multiplication. Since the calculations require dealing with 600-digit numbers, efficiency is obviously important. The straightforward algorithm for exponentiation would require about 10^{600} multiplications, whereas the algorithm presented requires only about 4,000 in the worst case.

Exercises

2.1 Order the following functions by growth rate: N, $\sqrt{N}$, $N^{1.5}$, N^2, $N \log N$, $N \log \log N$, $N \log^2 N$, $N \log(N^2)$, $2/N$, 2^N, $2^{N/2}$, 37, $N^2 \log N$, N^3. Indicate which functions grow at the same rate.

2.2 Suppose $T_1(N) = O(f(N))$ and $T_2(N) = O(f(N))$. Which of the following are true?
a. $T_1(N) + T_2(N) = O(f(N))$
b. $T_1(N) - T_2(N) = o(f(N))$
c. $\dfrac{T_1(N)}{T_2(N)} = O(1)$
d. $T_1(N) = O(T_2(N))$

2.3 Which function grows faster: $N \log N$ or $N^{1+\epsilon/\sqrt{\log N}}$, $\epsilon > 0$?

2.4 Prove that for any constant k, $\log^k N = o(N)$.

2.5 Find two functions $f(N)$ and $g(N)$ such that neither $f(N) = O(g(N))$ nor $g(N) = O(f(N))$.

2.6 In a recent court case, a judge cited a city for contempt and ordered a fine of \$2 for the first day. Each subsequent day, until the city followed the judge's order, the fine was squared (i.e., the fine progressed as follows: \$2, \$4, \$16, \$256, \$65,536, . . .).
a. What would be the fine on day N?
b. How many days would it take for the fine to reach D dollars (a Big-Oh answer will do)?

2.7 For each of the following six program fragments:
a. Give an analysis of the running time (Big-Oh will do).
b. Implement the code in the language of your choice, and give the running time for several values of N.
c. Compare your analysis with the actual running times.

```
(1)  sum = 0;
     for( i = 0; i < n; ++i )
         ++sum;
(2)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < n; ++j )
             ++sum;
(3)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < n * n; ++j )
             ++sum;
(4)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < i; ++j )
             ++sum;
(5)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < i * i; ++j )
             for( k = 0; k < j; ++k )
                 ++sum;
(6)  sum = 0;
     for( i = 1; i < n; ++i )
         for( j = 1; j < i * i; ++j )
             if( j % i == 0 )
                 for( k = 0; k < j; ++k )
                     ++sum;
```

2.8 Suppose you need to generate a *random* permutation of the first N integers. For example, {4, 3, 1, 5, 2} and {3, 1, 4, 2, 5} are legal permutations, but {5, 4, 1, 2, 1} is not, because one number (1) is duplicated and another (3) is missing. This routine is often used in simulation of algorithms. We assume the existence of a random number generator, `r`, with method `randInt(i,j)`, that generates integers between `i` and `j` with equal probability. Here are three algorithms:

1. Fill the array `a` from `a[0]` to `a[N-1]` as follows: To fill `a[i]`, generate random numbers until you get one that is not already in `a[0]`, `a[1]`, . . . , `a[i-1]`.
2. Same as algorithm (1), but keep an extra array called the `used` array. When a random number, `ran`, is first put in the array `a`, set `used[ran] = true`. This means that when filling `a[i]` with a random number, you can test in one step to see whether the random number has been used, instead of the (possibly) `i` steps in the first algorithm.
3. Fill the array such that `a[i] = i+1`. Then

```
for( i = 1; i < n; ++i )
    swap( a[ i ], a[ randInt( 0, i ) ] );
```

a. Prove that all three algorithms generate only legal permutations and that all permutations are equally likely.

b. Give as accurate (Big-Oh) an analysis as you can of the *expected* running time of each algorithm.
c. Write (separate) programs to execute each algorithm 10 times, to get a good average. Run program (1) for $N = 250, 500, 1{,}000, 2{,}000$; program (2) for $N = 25{,}000, 50{,}000, 100{,}000, 200{,}000, 400{,}000, 800{,}000$; and program (3) for $N = 100{,}000, 200{,}000, 400{,}000, 800{,}000, 1{,}600{,}000, 3{,}200{,}000, 6{,}400{,}000$.
d. Compare your analysis with the actual running times.
e. What is the worst-case running time of each algorithm?

2.9 Complete the table in Figure 2.2 with estimates for the running times that were too long to simulate. Interpolate the running times for these algorithms and estimate the time required to compute the maximum subsequence sum of 1 million numbers. What assumptions have you made?

2.10 Determine, for the typical algorithms that you use to perform calculations by hand, the running time to do the following:
a. Add two N-digit integers.
b. Multiply two N-digit integers.
c. Divide two N-digit integers.

2.11 An algorithm takes 0.5 ms for input size 100. How long will it take for input size 500 if the running time is the following (assume low-order terms are negligible)?
a. linear
b. $O(N \log N)$
c. quadratic
d. cubic

2.12 An algorithm takes 0.5 ms for input size 100. How large a problem can be solved in 1 min if the running time is the following (assume low-order terms are negligible)?
a. linear
b. $O(N \log N)$
c. quadratic
d. cubic

2.13 How much time is required to compute $f(x) = \sum_{i=0}^{N} a_i x^i$:
a. Using a simple routine to perform exponentiation?
b. Using the routine in Section 2.4.4?

2.14 Consider the following algorithm (known as *Horner's rule*) to evaluate $f(x) = \sum_{i=0}^{N} a_i x^i$:

```
poly = 0;
for( i = n; i >= 0; --i )
    poly = x * poly + a[i];
```

a. Show how the steps are performed by this algorithm for $x = 3, f(x) = 4x^4 + 8x^3 + x + 2$.
b. Explain why this algorithm works.
c. What is the running time of this algorithm?

2.15 Give an efficient algorithm to determine if there exists an integer i such that $A_i = i$ in an array of integers $A_1 < A_2 < A_3 < \cdots < A_N$. What is the running time of your algorithm?

2.16 Write an alternative `gcd` algorithm based on the following observations (arrange so that $a > b$):
a. $gcd(a, b) = 2gcd(a/2, b/2)$ if a and b are both even.
b. $gcd(a, b) = gcd(a/2, b)$ if a is even and b is odd.
c. $gcd(a, b) = gcd(a, b/2)$ if a is odd and b is even.
d. $gcd(a, b) = gcd((a + b)/2, (a - b)/2)$ if a and b are both odd.

2.17 Give efficient algorithms (along with running time analyses) to
a. Find the minimum subsequence sum.
* b. Find the minimum *positive* subsequence sum.
* c. Find the maximum subsequence *product*.

2.18 An important problem in numerical analysis is to find a solution to the equation $f(X) = 0$ for some arbitrary f. If the function is continuous and has two points *low* and *high* such that $f(low)$ and $f(high)$ have opposite signs, then a root must exist between *low* and *high* and can be found by a binary search. Write a function that takes as parameters f, *low,* and *high* and solves for a zero. What must you do to ensure termination?

2.19 The maximum contiguous subsequence sum algorithms in the text do not give any indication of the actual sequence. Modify them so that they return in a single object the value of the maximum subsequence and the indices of the actual sequence.

2.20 a. Write a program to determine if a positive integer, N, is prime.
b. In terms of N, what is the worst-case running time of your program? (You should be able to do this in $O(\sqrt{N})$.)
c. Let B equal the number of bits in the binary representation of N. What is the value of B?
d. In terms of B, what is the worst-case running time of your program?
e. Compare the running times to determine if a 20-bit number and a 40-bit number are prime.
f. Is it more reasonable to give the running time in terms of N or B? Why?

* **2.21** The Sieve of Eratosthenes is a method used to compute all primes less than N. We begin by making a table of integers 2 to N. We find the smallest integer, i, that is not crossed out, print i, and cross out i, $2i$, $3i$, ... When $i > \sqrt{N}$, the algorithm terminates. What is the running time of this algorithm?

2.22 Show that X^{62} can be computed with only eight multiplications.

2.23 Write the fast exponentiation routine without recursion.

2.24 Give a precise count on the number of multiplications used by the fast exponentiation routine. (*Hint:* Consider the binary representation of N.)

2.25 Programs A and B are analyzed and found to have worst-case running times no greater than $150N \log_2 N$ and N^2, respectively. Answer the following questions, if possible:

a. Which program has the better guarantee on the running time for large values of N ($N > 10{,}000$)?
b. Which program has the better guarantee on the running time for small values of N ($N < 100$)?
c. Which program will run faster *on average* for $N = 1{,}000$?
d. Is it possible that program B will run faster than program A on *all* possible inputs?

2.26 A majority element in an array, A, of size N is an element that appears more than $N/2$ times (thus, there is at most one). For example, the array

$$3, 3, 4, 2, 4, 4, 2, 4, 4$$

has a majority element (4), whereas the array

$$3, 3, 4, 2, 4, 4, 2, 4$$

does not. If there is no majority element, your program should indicate this. Here is a sketch of an algorithm to solve the problem:

First, a candidate *majority element is found (this is the harder part). This candidate is the only element that could possibly be the majority element. The second step determines if this candidate is actually the majority. This is just a sequential search through the array. To find a candidate in the array, A, form a second array, B. Then compare A_1 and A_2. If they are equal, add one of these to B; otherwise do nothing. Then compare A_3 and A_4. Again if they are equal, add one of these to B; otherwise do nothing. Continue in this fashion until the entire array is read. Then recursively find a candidate for B; this is the candidate for A (why?).*

a. How does the recursion terminate?
* b. How is the case where N is odd handled?
* c. What is the running time of the algorithm?
d. How can we avoid using an extra array, B?
* e. Write a program to compute the majority element.

2.27 The input is an N by N matrix of numbers that is already in memory. Each individual row is increasing from left to right. Each individual column is increasing from top to bottom. Give an $O(N)$ worst-case algorithm that decides if a number X is in the matrix.

2.28 Design efficient algorithms that take an array of positive numbers `a`, and determine:
a. the maximum value of `a[j]+a[i]`, with `j` $\geq$ `i`.
b. the maximum value of `a[j]-a[i]`, with `j` $\geq$ `i`.
c. the maximum value of `a[j]*a[i]`, with `j` $\geq$ `i`.
d. the maximum value of `a[j]/a[i]`, with `j` $\geq$ `i`.

* **2.29** Why is it important to assume that integers in our computer model have a fixed size?

2.30 Consider the word puzzle problem on page 2. Suppose we fix the size of the longest word to be 10 characters.

a. In terms of R and C, which are the number of rows and columns in the puzzle, and W, which is the number of words, what are the running times of the algorithms described in Chapter 1?
b. Suppose the word list is presorted. Show how to use binary search to obtain an algorithm with significantly better running time.

2.31 Suppose that line 15 in the binary search routine had the statement `low = mid` instead of `low = mid + 1`. Would the routine still work?

2.32 Implement the binary search so that only one two-way comparison is performed in each iteration.

2.33 Suppose that lines 15 and 16 in algorithm 3 (Fig. 2.7) are replaced by

```
15        int maxLeftSum  = maxSumRec( a, left, center - 1 );
16        int maxRightSum = maxSumRec( a, center, right );
```

Would the routine still work?

***2.34** The inner loop of the cubic maximum subsequence sum algorithm performs $N(N+1)(N+2)/6$ iterations of the innermost code. The quadratic version performs $N(N+1)/2$ iterations. The linear version performs N iterations. What pattern is evident? Can you give a combinatoric explanation of this phenomenon?

References

Analysis of the running time of algorithms was first made popular by Knuth in the three-part series [5], [6], and [7]. Analysis of the *gcd* algorithm appears in [6]. Another early text on the subject is [1].

Big-Oh, big-omega, big-theta, and little-oh notation were advocated by Knuth in [8]. There is still no uniform agreement on the matter, especially when it comes to using $\Theta()$. Many people prefer to use $O()$, even though it is less expressive. Additionally, $O()$ is still used in some corners to express a lower bound, when $\Omega()$ is called for.

The maximum subsequence sum problem is from [3]. The series of books [2], [3], and [4] show how to optimize programs for speed.

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.
2. J. L. Bentley, *Writing Efficient Programs,* Prentice Hall, Englewood Cliffs, N.J., 1982.
3. J. L. Bentley, *Programming Pearls,* Addison-Wesley, Reading, Mass., 1986.
4. J. L. Bentley, *More Programming Pearls,* Addison-Wesley, Reading, Mass., 1988.
5. D. E. Knuth, *The Art of Computer Programming, Vol 1: Fundamental Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1997.
6. D. E. Knuth, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1998.
7. D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching,* 2d ed., Addison-Wesley, Reading, Mass., 1998.
8. D. E. Knuth, "Big Omicron and Big Omega and Big Theta," *ACM SIGACT News,* 8 (1976), 18–23.

CHAPTER 3

Lists, Stacks, and Queues

This chapter discusses three of the most simple and basic data structures. Virtually every significant program will use at least one of these structures explicitly, and a stack is always implicitly used in a program, whether or not you declare one. Among the highlights of this chapter, we will . . .

- Introduce the concept of Abstract Data Types (ADTs).
- Show how to efficiently perform operations on lists.
- Introduce the stack ADT and its use in implementing recursion.
- Introduce the queue ADT and its use in operating systems and algorithm design.

In this chapter, we provide code that implements a significant subset of two library classes: `vector` and `list`.

3.1 Abstract Data Types (ADTs)

An **abstract data type** (ADT) is a set of objects together with a set of operations. Abstract data types are mathematical abstractions; nowhere in an ADT's definition is there any mention of *how* the set of operations is implemented. Objects such as lists, sets, and graphs, along with their operations, can be viewed as ADTs, just as integers, reals, and booleans are data types. Integers, reals, and booleans have operations associated with them, and so do ADTs. For the set ADT, we might have such operations as *add, remove, size,* and *contains.* Alternatively, we might only want the two operations *union* and *find,* which would define a different ADT on the set.

The C++ class allows for the implementation of ADTs, with appropriate hiding of implementation details. Thus, any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate method. If for some reason implementation details need to be changed, it should be easy to do so by merely changing the routines that perform the ADT operations. This change, in a perfect world, would be completely transparent to the rest of the program.

There is no rule telling us which operations must be supported for each ADT; this is a design decision. Error handling and tie breaking (where appropriate) are also generally up to the program designer. The three data structures that we will study in this chapter are primary examples of ADTs. We will see how each can be implemented in several ways, but