Recursion

5

5.1 Recursive Definitions

One of the basic rules for defining new objects or concepts is that the definition should contain only such terms that have already been defined or that are obvious. Therefore, an object that is defined in terms of itself is a serious violation of this rule—a vicious circle. On the other hand, there are many programming concepts that define themselves. As it turns out, formal restrictions imposed on definitions such as existence and uniqueness are satisfied and no violation of the rules takes place. Such definitions are called *recursive definitions,* and are used primarily to define infinite sets. When defining such a set, giving a complete list of elements is impossible, and for large finite sets, it is inefficient. Thus, a more efficient way has to be devised to determine if an object belongs to a set.

A recursive definition consists of two parts. In the first part, called the *anchor* or the *ground case,* the basic elements that are the building blocks of all other elements of the set are listed. In the second part, rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed. These rules are applied again and again to generate new objects. For example, to construct the set of natural numbers, one basic element, 0, is singled out, and the operation of incrementing by 1 is given as:

1. $0 \in \mathbf{N}$;
2. if $n \in \mathbf{N}$, then $(n + 1) \in \mathbf{N}$;
3. there are no other objects in the set **N**.

(More axioms are needed to ensure that only the set that we know as the natural numbers can be constructed by these rules.)

According to these rules, the set of natural numbers **N** consists of the following items: 0, 0 + 1, 0 + 1 + 1, 0 + 1 + 1 + 1, and so on. Although the set **N** contains objects (and only such objects) that we call natural numbers, the definition results in a somewhat unwieldy list of elements. Can you imagine doing arithmetic on large numbers

using such a specification? Therefore, it is more convenient to use the following definition, which encompasses the whole range of Arabic numeric heritage:

1. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ∈ **N**;
2. if $n \in \mathbf{N}$, then $n0, n1, n2, n3, n4, n5, n6, n7, n8, n9 \in \mathbf{N}$;
3. these are the only natural numbers.

Then the set **N** includes all possible combinations of the basic building blocks 0 through 9.

Recursive definitions serve two purposes: *generating* new elements, as already indicated, and *testing* whether an element belongs to a set. In the case of testing, the problem is solved by reducing it to a simpler problem, and if the simpler problem is still too complex it is reduced to an even simpler problem, and so on, until it is reduced to a problem indicated in the anchor. For instance, is 123 a natural number? According to the second condition of the definition introducing the set **N**, $123 \in \mathbf{N}$ if $12 \in \mathbf{N}$ and the first condition already says that $3 \in \mathbf{N}$; but $12 \in \mathbf{N}$ if $1 \in \mathbf{N}$ and $2 \in \mathbf{N}$, and they both belong to **N**.

The ability to decompose a problem into simpler subproblems of the same kind is sometimes a real blessing, as we shall see in the discussion of quicksort in Section 9.3.3, or a curse, as we shall see shortly in this chapter.

Recursive definitions are frequently used to define functions and sequences of numbers. For instance, the factorial function, !, can be defined in the following manner:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ (anchor)} \\ n \cdot (n-1)! & \text{if } n > 0 \text{ (inductive step)} \end{cases}$$

Using this definition, we can generate the sequence of numbers

$$1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, \ldots$$

which includes the factorials of the numbers $0, 1, 2, \ldots, 10, \ldots$

Another example is the definition

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ f(n-1) + \dfrac{1}{f(n-1)} & \text{if } n > 0 \end{cases}$$

which generates the sequence of rational numbers

$$1, 2, \frac{5}{2}, \frac{29}{10}, \frac{941}{290}, \frac{969{,}581}{272{,}890}, \ldots$$

Recursive definitions of sequences have one undesirable feature: to determine the value of an element s_n of a sequence, we first have to compute the values of some or all of the previous elements, $s_1, \ldots, s_{n-1}$. For example, calculating the value of 3! requires us to first compute the values of 0!, 1!, and 2!. Computationally, this is undesirable because it forces us to make calculations in a roundabout way. Therefore, we want to find an equivalent definition or formula that makes no references to other elements of the sequence. Generally, finding such a formula is a difficult problem that cannot always be solved. But the formula is preferable to a recursive definition because it

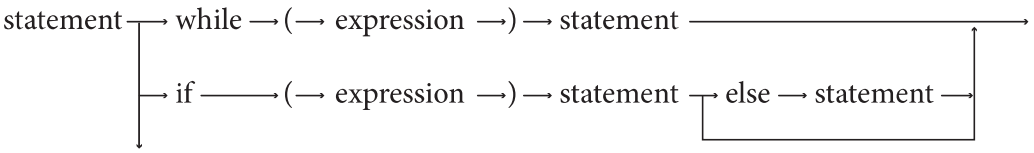
simplifies the computational process and allows us to find the answer for an integer n without computing the values for integers $0, 1, \ldots, n-1$. For example, a definition of the sequence g,

$$g(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 \cdot g(n-1) & \text{if } n > 0 \end{cases}$$

can be converted into the simple formula

$$g(n) = 2^n$$

In the foregoing discussion, recursive definitions have been dealt with only theoretically, as a definition used in mathematics. Naturally, our interest is in computer science. One area where recursive definitions are used extensively is in the specification of the grammars of programming languages. Every programming language manual contains—either as an appendix or throughout the text—a specification of all valid language elements. Grammar is specified either in terms of block diagrams or in terms of the Backus-Naur form (BNF). For example, the syntactic definition of a statement in the C++ language can be presented in the block diagram form:



or in BNF:

```
<statement> ::= while (<expression>) <statement> |
                if (<expression>) <statement> |
                 if (<expression>) <statement>else<statement> |
                ...
```

The language element <statement> is defined recursively, in terms of itself. Such definitions naturally express the possibility of creating such syntactic constructs as nested statements or expressions.

Recursive definitions are also used in programming. The good news is that virtually no effort is needed to make the transition from a recursive definition of a function to its implementation in C++. We simply make a translation from the formal definition into C++ syntax. Hence, for example, a C++ equivalent of factorial is the function

```
unsigned int factorial (unsigned int n) {
  if (n == 0)
       return 1;
  else return n * factorial (n - 1);
}
```

The problem now seems to be more critical because it is far from clear how a function calling itself can possibly work, let alone return the correct result. This chapter shows that it is possible for such a function to work properly. Recursive definitions on most computers are eventually implemented using a run-time stack, although

the whole work of implementing recursion is done by the operating system, and the source code includes no indication of how it is performed. E. W. Dijkstra introduced the idea of using a stack to implement recursion. To better understand recursion and to see how it works, it is necessary to discuss the processing of function calls and to look at operations carried out by the system at function invocation and function exit.

5.2 Function Calls and Recursion Implementation

What happens when a function is called? If the function has formal parameters, they have to be initialized to the values passed as actual parameters. In addition, the system has to know where to resume execution of the program after the function has finished. The function can be called by other functions or by the main program (the function `main()`). The information indicating where it has been called from has to be remembered by the system. This could be done by storing the return address in main memory in a place set aside for return addresses, but we do not know in advance how much space might be needed, and allocating too much space for that purpose alone is not efficient.

For a function call, more information has to be stored than just a return address. Therefore, dynamic allocation using the run-time stack is a much better solution. But what information should be preserved when a function is called? First, local variables must be stored. If function `f1()`, which contains a declaration of a local variable `x`, calls function `f2()`, which locally declares the variable `x`, the system has to make a distinction between these two variables `x`. If `f2()` uses a variable `x`, then its own `x` is meant; if `f2()` assigns a value to `x`, then `x` belonging to `f1()` should be left unchanged. When `f2()` is finished, `f1()` can use the value assigned to its private `x` before `f2()` was called. This is especially important in the context of the present chapter, when `f1()` is the same as `f2()`, when a function calls itself recursively. How does the system make a distinction between these two variables `x`?

The state of each function, including `main()`, is characterized by the contents of all local variables, by the values of the function's parameters, and by the return address indicating where to restart in the calling function. The data area containing all this information is called an *activation record* or a *stack frame* and is allocated on the run-time stack. An activation record exists for as long as a function owning it is executing. This record is a private pool of information for the function, a repository that stores all information necessary for its proper execution and how to return to where it was called from. Activation records usually have a short life span because they are dynamically allocated at function entry and deallocated upon exiting. Only the activation record of `main()` outlives every other activation record.

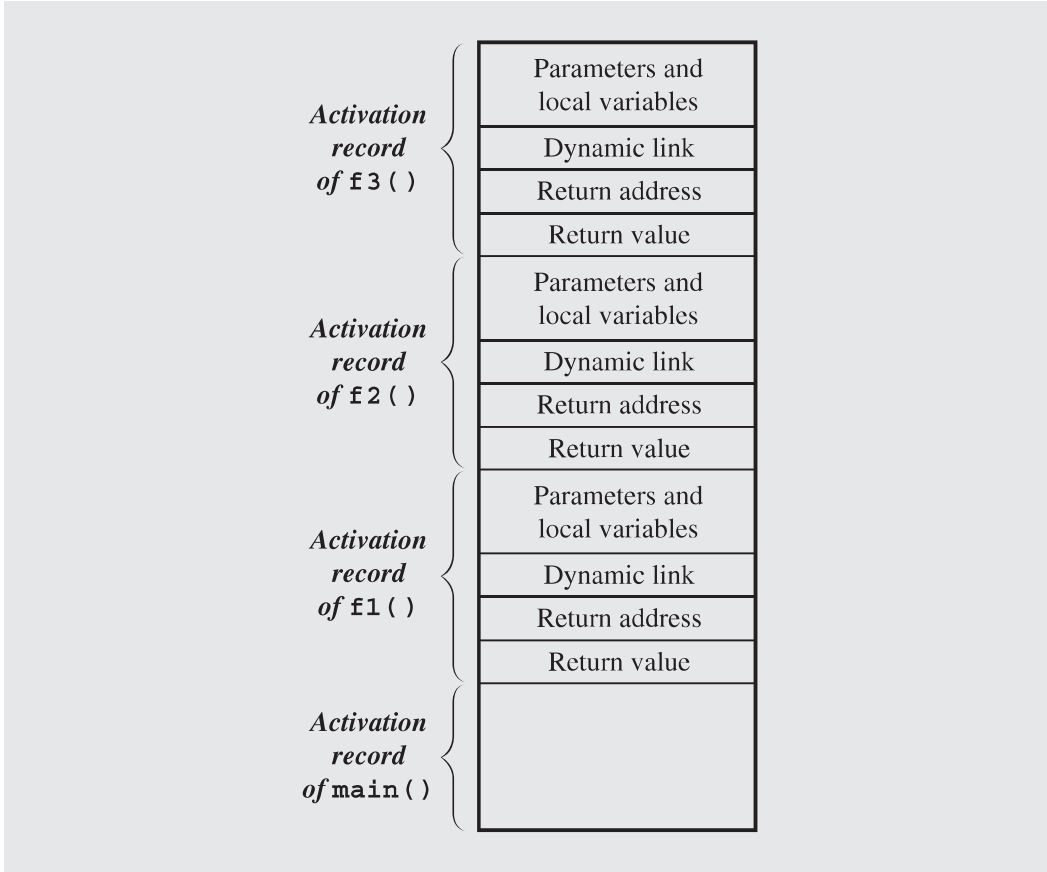
An activation record usually contains the following information:

- Values for all parameters to the function, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items.
- Local variables that can be stored elsewhere, in which case, the activation record contains only their descriptors and pointers to the locations where they are stored.

- The return address to resume control by the caller, the address of the caller's instruction immediately following the call.
- A dynamic link, which is a pointer to the caller's activation record.
- The returned value for a function not declared as `void`. Because the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller.

As mentioned, if a function is called either by `main()` or by another function, then its activation record is created on the run-time stack. The run-time stack always reflects the current state of the function. For example, suppose that `main()` calls function `f1()`, `f1()` calls `f2()`, and `f2()` in turn calls `f3()`. If `f3()` is being executed, then the state of the run-time stack is as shown in Figure 5.1. By the nature of the stack, if the activation record for `f3()` is popped by moving the stack pointer right below the return value of `f3()`, then `f2()` resumes execution and now has free access to the private pool of information necessary for reactivation of its execution. On the other hand, if `f3()` happens to call another function `f4()`, then the run-time stack increases its height because the activation record for `f4()` is created on the stack and the activity of `f3()` is suspended.

FIGURE 5.1 Contents of the run-time stack when `main()` calls function `f1()`, `f1()` calls `f2()`, and `f2()` calls `f3()`.



Creating an activation record whenever a function is called allows the system to handle recursion properly. Recursion is calling a function that happens to have the same name as the caller. Therefore, a recursive call is not literally a function calling itself, but rather an instantiation of a function calling another instantiation of the same original. These invocations are represented internally by different activation records and are thus differentiated by the system.

5.3 Anatomy of a Recursive Call

The function that defines raising any number x to a nonnegative integer power n is a good example of a recursive function. The most natural definition of this function is given by:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

A C++ function for computing x^n can be written directly from the definition of a power:

```
/* 102 */ double power (double x, unsigned int n) {
/* 103 */      if (n == 0)
/* 104 */           return 1.0;
           // else
/* 105 */          return x * power(x,n-1);
          }
```

Using this definition, the value of x^4 can be computed in the following way:

$$\begin{aligned} x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) = x \cdot (x \cdot (x \cdot x^1)) = x \cdot (x \cdot (x \cdot (x \cdot x^0))) \\ = x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x)) \\ = x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x \end{aligned}$$

The repetitive application of the inductive step eventually leads to the anchor, which is the last step in the chain of recursive calls. The anchor produces 1 as a result of raising x to the power of zero; the result is passed back to the previous recursive call. Now, that call, whose execution has been pending, returns its result, $x \cdot 1 = x$. The third call, which has been waiting for this result, computes its own result, namely, $x \cdot x$, and returns it. Next, this number $x \cdot x$ is received by the second call, which multiplies it by x and returns the result, $x \cdot x \cdot x$, to the first invocation of `power()`. This call receives $x \cdot x \cdot x$, multiplies it by x, and returns the final result. In this way, each new call increases the level of recursion, as follows:

call 1	$x^4 = x \cdot x^3 = x \cdot x \cdot x \cdot x$
call 2	$x \cdot x^2 = x \cdot x \cdot x$
call 3	$x \cdot x^1 = x \cdot x$
call 4	$x \cdot x^0 = x \cdot 1 = x$
call 5	1

or alternatively, as

```
call 1          power(x,4)
call 2                power(x,3)
call 3                      power(x,2)
call 4                            power(x,1)
call 5                                  power(x,0)
call 5                                  1
call 4                            x
call 3                      x · x
call 2                x · x · x
call 1          x · x · x · x
```

What does the system do as the function is being executed? As we already know, the system keeps track of all calls on its run-time stack. Each line of code is assigned a number by the system,[1] and if a line is a function call, then its number is a return address. The address is used by the system to remember where to resume execution after the function has completed. For this example, assume that the lines in the function `power()` are assigned the numbers 102 through 105 and that it is called `main()` from the statement

```
          int main()
          { ...
/* 136 */  y = power(5.6,2);
            ...
          }
```

A trace of the recursive calls is relatively simple, as indicated by this diagram

```
call 1          power(5.6,2)
call 2                power(5.6,1)
call 3                      power(5.6,0)
call 3                      1
call 2                5.6
call 1          31.36
```

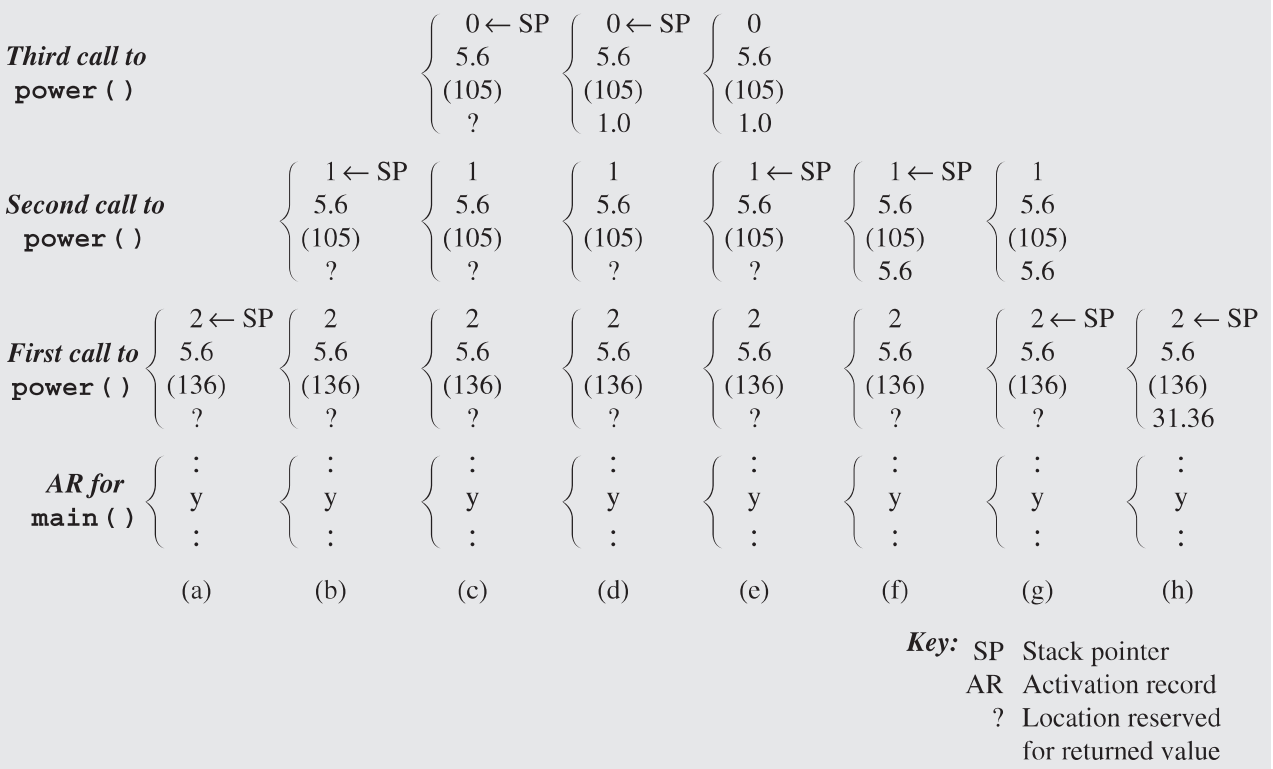
because most of the operations are performed on the run-time stack.

When the function is invoked for the first time, four items are pushed onto the run-time stack: the return address 136, the actual parameters 5.6 and 2, and one location reserved for the value returned by `power()`. Figure 5.2a represents this situation. (In this and subsequent diagrams, SP is a stack pointer, AR is an activation record, and question marks stand for locations reserved for the returned values. To distinguish values from addresses, the latter are parenthesized, although addresses are numbers exactly like function arguments.)

Now the function `power()` is executed. First, the value of the second argument, 2, is checked, and `power()` tries to return the value of 5.6 · `power(5.6,1)` because

[1]This is not quite precise because the system uses machine code rather than source code to execute programs. This means that one line of source program is usually implemented by several machine instructions.

FIGURE 5.2 Changes to the run-time stack during execution of `power(5.6,2)`.



that argument is not 0. This cannot be done immediately because the system does not know the value of `power(5.6,1)`; it must be computed first. Therefore, `power()` is called again with the arguments 5.6 and 1. But before this call is executed, the run-time stack receives new items, and its contents are shown in Figure 5.2b.

Again, the second argument is checked to see if it is 0. Because it is equal to 1, `power()` is called for the third time, this time with the arguments 5.6 and 0. Before the function is executed, the system remembers the arguments and the return address by putting them on the stack, not forgetting to allocate one cell for the result. Figure 5.2c contains the new contents of the stack.

Again, the question arises: is the second argument equal to zero? Because it finally is, a concrete value—namely, 1.0—can be returned and placed on the stack, and the function is finished without making any additional calls. At this point, there are two pending calls on the run-time stack—the calls to `power()`—that have to be completed. How is this done? The system first eliminates the activation record of `power()` that has just finished. This is performed logically by popping all its fields (the result, two arguments, and the return address) off the stack. We say "logically" because physically all these fields remain on the stack and only the SP is decremented appropriately. This is important because we do not want the result to be destroyed since it has not been used yet. Before and after completion of the last call of `power()`, the stack looks the same, but the SP's value is changed (see Figures 5.2d and 5.2e).

Now the second call to `power()` can complete because it waited for the result of the call `power(5.6,0)`. This result, 1.0, is multiplied by 5.6 and stored in the field allocated for the result. After that, the system can pop the current activation record off the stack by decrementing the SP, and it can finish the execution of the first call to `power()` that needed the result for the second call. Figure 5.2f shows the contents of the stack before changing the SP's value, and Figure 5.2g shows the contents of the stack after this change. At this moment, `power()` can finish its first call by multiplying the result of its second call, 5.6, by its first argument, also 5.6. The system now returns to the function that invoked `power()`, and the final value, 31.36, is assigned to `y`. Right before the assignment is executed, the content of the stack looks like Figure 5.2h.

The function `power()` can be implemented differently, without using any recursion, as in the following loop:

```
double nonRecPower(double x, unsigned int n) {
  double result = 1;
  for (result = x; n > 1; --n)
       result *= x;
  return result;
}
```

Do we gain anything by using recursion instead of a loop? The recursive version seems to be more intuitive because it is similar to the original definition of the power function. The definition is simply expressed in C++ without losing the original structure of the definition. The recursive version increases program readability, improves self-documentation, and simplifies coding. In our example, the code of the nonrecursive version is not substantially larger than in the recursive version, but for most recursive implementations, the code is shorter than it is in the nonrecursive implementations.

5.4 Tail Recursion

All recursive definitions contain a reference to a set or function being defined. There are, however, a variety of ways such a reference can be implemented. This reference can be done in a straightforward manner or in an intricate fashion, just once or many times. There may be many possible levels of recursion or different levels of complexity. In the following sections, some of these types are discussed, starting with the simplest case, *tail recursion*.

Tail recursion is characterized by the use of only one recursive call at the very end of a function implementation. In other words, when the call is made, there are no statements left to be executed by the function; the recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect. For example, the function `tail()` defined as

```
void tail(int i) {
  if (i > 0) {
    cout << i << '';
    tail(i-1);
  }
}
```

is an example of a function with tail recursion, whereas the function `nonTail()` defined as

```
void nonTail(int i) {
  if (i > 0) {
    nonTail(i-1);
    cout << i << '';
    nonTail(i-1);
  }
}
```

is not. Tail recursion is simply a glorified loop and can be easily replaced by one. In this example, it is replaced by substituting a loop for the `if` statement and decrementing the variable `i` in accordance with the level of recursive call. In this way, `tail()` can be expressed by an iterative function:

```
void iterativeEquivalentOfTail(int i) {
  for ( ; i > 0; i--)
    cout << i << '';

}
```

Is there any advantage in using tail recursion over iteration? For languages such as C++, there may be no compelling advantage, but in a language such as Prolog, which has no explicit loop construct (loops are simulated by recursion), tail recursion acquires a much greater weight. In languages endowed with a loop or its equivalents, such as an `if` statement combined with a `goto` statement, tail recursion should not be used.

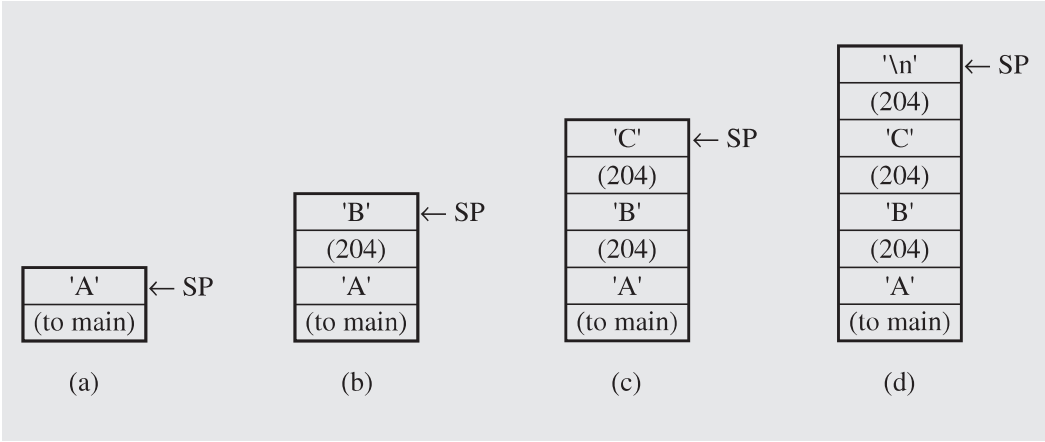
5.5 Nontail Recursion

Another problem that can be implemented in recursion is printing an input line in reverse order. Here is a simple recursive implementation:

```
/* 200 */ void reverse() {
                char ch;
/* 201 */       cin.get(ch);
/* 202 */       if (ch != '\n') {
/* 203 */           reverse();
/* 204 */           cout.put(ch);
                }
          }
```

Where is the trick? It does not seem possible that the function does anything. But it turns out that, by the power of recursion, it does exactly what it was designed for. `main()` calls `reverse()` and the input is the string: "ABC." First, an activation record is created with cells for the variable `ch` and the return address. There is no need to reserve a cell for a result, because no value is returned, which is indicated by using `void` in front of the function's name. The function `get()` reads in the first character, "A." Figure 5.3a shows the contents of the run-time stack right before `reverse()` calls itself recursively for the first time.

FIGURE 5.3 Changes on the run-time stack during the execution of `reverse()`.



The second character is read in and checked to see if it is the end-of-line character, and if not, `reverse()` is called again. But in either case, the value of `ch` is pushed onto the run-time stack along with the return address. Before `reverse()` is called for a third time (the second time recursively), there are two more items on the stack (see Figure 5.3b).

Note that the function is called as many times as the number of characters contained in the input string, including the end-of-line character. In our example, `reverse()` is called four times, and the run-time stack during the last call is shown in Figure 5.3d.

On the fourth call, `get()` finds the end-of-line character and `reverse()` executes no other statement. The system retrieves the return address from the activation record and discards this record by decrementing SP by the proper number of bytes. Execution resumes from line 204, which is a print statement. Because the activation record of the third call is now active, the value of `ch`, the letter "C," is output as the first character. Next, the activation record of the third call to `reverse()` is discarded and now SP points to where "B" is stored. The second call is about to be finished, but first, "B" is assigned to `ch` and then the statement on line 204 is executed, which results in printing "B" on the screen right after "C." Finally, the activation record of the first call to `reverse()` is reached. Then "A" is printed, and what can be seen on the screen is the string "CBA." The first call is finally finished and the program continues execution in `main()`.

Compare the recursive implementation with a nonrecursive version of the same function:

```
void simpleIterativeReverse() {
  char stack[80];
  register int top = 0;
  cin.getline(stack,80);
  for (top = strlen(stack) - 1; top >= 0; cout.put(stack[top--]));
}
```

The function is quite short and, perhaps, a bit more cryptic than its recursive counterpart. What is the difference then? Keep in mind that the brevity and relative simplicity of the second version are due mainly to the fact that we want to reverse

a string or array of characters. This means that functions like `strlen()` and `getline()` from the standard C++ library can be used. If we are not supplied with such functions, then our iterative function has to be implemented differently:

```
void iterativeReverse() {
  char stack[80];
  register int top = 0;
  cin.get(stack[top]);
  while(stack[top]!='\n')
    cin.get(stack[++top]);
  for (top -= 2; top >= 0; cout.put(stack[top--]));
}
```

The `while` loop replaces `getline()` and the autoincrement of variable `top` replaces `strlen()`. The `for` loop is about the same as before. This discussion is not purely theoretical because reversing an input line consisting of integers uses the same implementation as `iterativeReverse()` after changing the data type of `stack` from `char` to `int` and modifying the `while` loop.

Note that the variable name `stack` used for the array is not accidental. We are just making explicit what is done implicitly by the system. Our stack takes over the run-time stack's duty. Its use is necessary here because one simple loop does not suffice, as in the case of tail recursion. In addition, the statement `put()` from the recursive version has to be accounted for. Note also that the variable `stack` is local to the function `iterativeReverse()`. However, if it were a requirement to have a global stack object `st`, then this implementation can be written as

```
void nonRecursiveReverse() {
   int ch;
   cin.get(ch);
   while (ch != '\n') {
       st.push(ch);
       cin.get(ch);
   }
   while (!st.empty())
       cout.put(st.pop());
}
```

with the declaration `Stack<char> st` outside the function.

After comparing `iterativeReverse()` to `nonRecursiveReverse()`, we can conclude that the first version is better because it is faster, no function calls are made, and the function is self-sufficient, whereas `nonRecursiveReverse()` calls at least one function during each loop iteration, slowing down execution.

One way or the other, the transformation of nontail recursion into iteration usually involves the explicit handling of a stack. Furthermore, when converting a function from a recursive into an iterative version, program clarity can be diminished and the brevity of program formulation lost. Iterative versions of recursive C++ functions are not as verbose as in other programming languages, so program brevity may not be an issue.

To conclude this section, consider a construction of the von Koch snowflake. The curve was constructed in 1904 by Swedish mathematician Helge von Koch as an example of a continuous and nondifferentiable curve with an infinite length and yet encompassing a finite area. Such a curve is a limit of an infinite sequence of snowflakes, of which the first three are presented in Figure 5.4. As in real snowflakes, two of these curves have six petals, but to facilitate the algorithm, it is treated as a combination of three identical curves drawn in different angles and joined together. One such curve is drawn in the following fashion:

1. Divide an interval *side* into three even parts.
2. Move one-third of *side* in the direction specified by *angle.*
3. Turn to the right 60° (i.e., turn –60°) and go forward one-third of *side.*
4. Turn to the left 120° and proceed forward one-third of *side.*
5. Turn right 60° and again draw a line one-third of *side* long.

The result of these five steps is summarized in Figure 5.5. This line, however, becomes more jagged if every one of the four intervals became a miniature of the whole curve; that is, if the process of drawing four lines were made for each of these *side*/3 long intervals. As a result, 16 intervals *side*/9 long would be drawn. The process may be continued indefinitely—at least in theory. Computer graphics resolution prevents us from going too far because if lines are smaller than the diameter of a pixel, we just see one dot on the screen.

FIGURE 5.4 Examples of von Koch snowflakes.

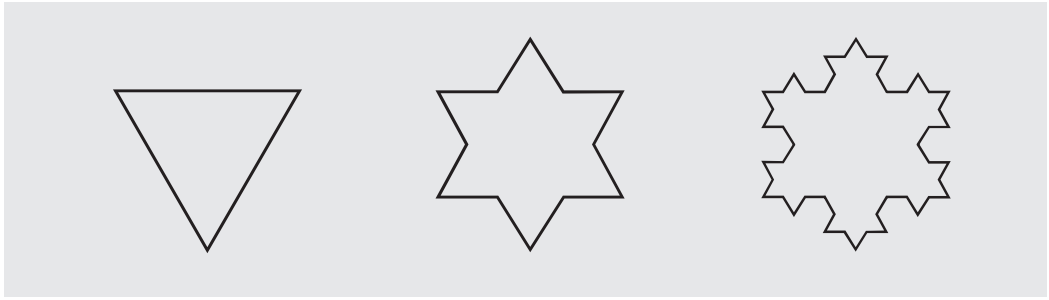
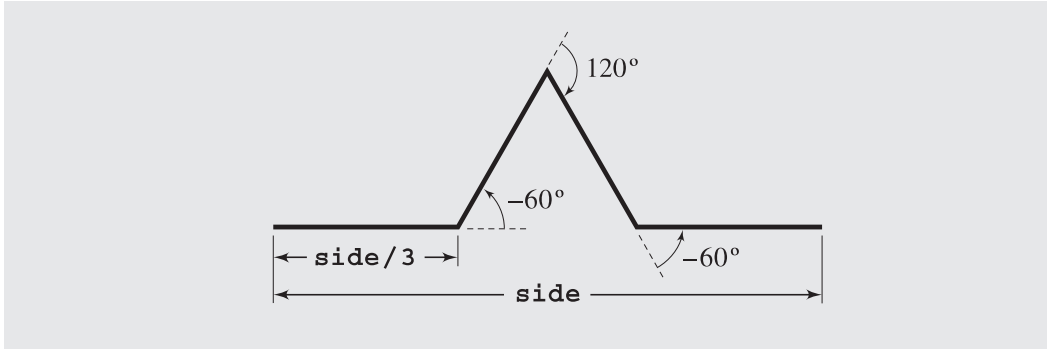


FIGURE 5.5 The process of drawing four sides of one segment of the von Koch snowflake.



The five steps that, instead of drawing one line of length *side,* draw four lines each of length one-third of *side* form one cycle only. Each of these four lines can also be compound lines drawn by the use of the described cycle. This is a situation in which recursion is well suited, which is reflected by the following pseudocode:

```
drawFourLines (side, level)
   if (level == 0)
     draw a line;
   else
     drawFourLines(side/3, level-1);
     turn left 60°;
     drawFourLines(side/3, level-1);
     turn right 120°;
     drawFourLines(side/3, level-1);
     turn left 60°;
     drawFourLines(side/3, level-1);
```

This pseudocode can be rendered almost without change into C++ code (Figure 5.6).

FIGURE 5.6 Recursive implementation of the von Koch snowflake.

```
//************************  vonKoch.h  ***************************
//                       Visual C++ program
//           A header file in a project of type MFC Application.

#define _USE_MATH_DEFINES
#include <cmath>

class vonKoch {
public:
    vonKoch(int,int,CDC*);
    void snowflake();
private:
    double side, angle;
    int level;
    CPoint currPt, pt;
    CDC *pen;
    void right(double x) {
        angle += x;
    }
    void left (double x) {
        angle -= x;
    }
    void drawFourLines(double side, int level);
};
```

FIGURE 5.6 (*continued*)

```
vonKoch::vonKoch(int s, int lvl, CDC *pDC) {
    pen = pDC;
    currPt.x = 200;
    currPt.y = 100;
    pen->MoveTo(currPt);
    angle = 0.0;
    side = s;
    level = lvl;
}

void vonKoch::drawFourLines(double side, int level) {
    // arguments to sin() and cos() are angles
    // specified in radians, i.e., the coefficient
    // PI/180 is necessary;
    if (level == 0) {
         pt.x = int(cos(angle*M_PI/180)*side) + currPt.x;
         pt.y = int(sin(angle*M_PI/180)*side) + currPt.y;
         pen->LineTo(pt);
         currPt.x = pt.x;
         currPt.y = pt.y;
    }
    else {
         drawFourLines(side/3,level-1);
         left (60);
         drawFourLines(side/3,level-1);
         right(120);
         drawFourLines(side/3,level-1);
         left (60);
         drawFourLines(side/3,level-1);
    }
}

void vonKoch::snowflake() {
    for (int i = 1; i <= 3; i++) {
        drawFourLines(side,level);
            right(120);
    }
}

// The function OnDraw() is generated by Visual C++ in snowflakeView.cpp
// when creating a snowflake project of type MFC Application;

#include "vonKoch.h"
```

Continues

FIGURE 5.6 (*continued*)

```
void CSnowflakeView::OnDraw(CDC* pDC)
{
    CSnowflakeDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
          return;

    // TODO: add draw code for native data here

    vonKoch(200,4,pDC).snowflake();
}
```

5.6 Indirect Recursion

The preceding sections discussed only direct recursion, where a function `f()` called itself. However, `f()` can call itself indirectly via a chain of other calls. For example, `f()` can call `g()`, and `g()` can call `f()`. This is the simplest case of indirect recursion.

The chain of intermediate calls can be of an arbitrary length, as in:

```
f() -> f1() -> f2() -> ··· -> fn() -> f()
```

There is also the situation when `f()` can call itself indirectly through different chains. Thus, in addition to the chain just given, another chain might also be possible. For instance

```
f() -> g1() -> g2() -> ··· -> gm() -> f()
```

This situation can be exemplified by three functions used for decoding information. `receive()` stores the incoming information in a buffer, `decode()` converts it into legible form, and `store()` stores it in a file. `receive()` fills the buffer and calls `decode()`, which in turn, after finishing its job, submits the buffer with decoded information to `store()`. After `store()` accomplishes its tasks, it calls `receive()` to intercept more encoded information using the same buffer. Therefore, we have the chain of calls

```
receive() -> decode() -> store() -> receive() -> decode() -> ···
```

which is finished when no new information arrives. These three functions work in the following manner:

```
receive(buffer)
   while buffer is not filled up
      if information is still incoming
         get a character and store it in buffer;
      else exit();
      decode(buffer);
```

```
decode(buffer)
    decode information in buffer;
    store(buffer);

store(buffer)
    transfer information from buffer to file;
    receive(buffer);
```

A more mathematically oriented example concerns formulas calculating the trigonometric functions sine, cosine, and tangent:

$$\sin(x) = \sin\left(\frac{x}{3}\right) \cdot \frac{(3 - \tan^2(\frac{x}{3}))}{(1 + \tan^2(\frac{x}{3}))}$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

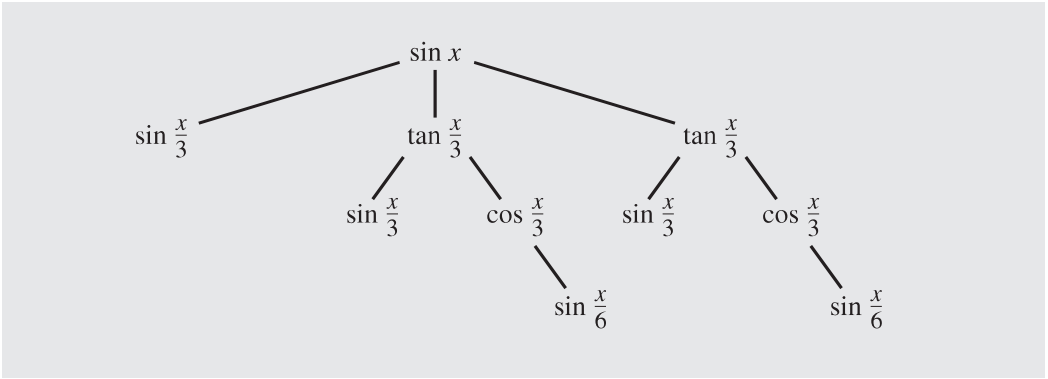
$$\cos(x) = 1 - \sin\left(\frac{x}{2}\right)$$

As usual in the case of recursion, there has to be an anchor in order to avoid falling into an infinite loop of recursive calls. In the case of sine, we can use the following approximation:

$$\sin(x) \approx x - \frac{x^3}{6}$$

where small values of x give a better approximation. To compute the sine of a number x such that its absolute value is greater than an assumed tolerance, we have to compute $\sin(\frac{x}{3})$ directly, $\sin(\frac{x}{3})$ indirectly through tangent, and also indirectly, $\sin(\frac{x}{6})$ through tangent and cosine. If the absolute value of $\frac{x}{3}$ is sufficiently small, which does not require other recursive calls, we can represent all the calls as a tree, as in Figure 5.7.

FIGURE 5.7 A tree of recursive calls for sin (x).



5.7 Nested Recursion

A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting:

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ h(2 + h(n)) & \text{if } n \leq 4 \end{cases}$$

Function h has a solution for all $n \geq 0$. This fact is obvious for all $n > 4$ and $n = 0$, but it has to be proven for $n = 1, 2, 3$, and 4. Thus, $h(2) = h(2 + h(4)) = h(2 + h(2 + h(8))) = 12$. (What are the values of $h(n)$ for $n = 1, 3$, and 4?)

Another example of nested recursion is a very important function originally suggested by Wilhelm Ackermann in 1928 and later modified by Rozsa Peter:

$$A(n,m) = \begin{cases} m+1 & \text{if } n = 0 \\ A(n-1,1) & \text{if } n > 0, m = 0 \\ A(n-1,A(n,m-1)) & \text{otherwise} \end{cases}$$

This function is interesting because of its remarkably rapid growth. It grows so fast that it is guaranteed not to have a representation by a formula that uses arithmetic operations such as addition, multiplication, and exponentiation. To illustrate the rate of growth of the Ackermann function, we need only show that

$$A(3,m) = 2^{m+3} - 3$$

$$A(4,m) = 2^{2^{2^{\cdot^{\cdot^{2^{16}}}}}} - 3$$

with a stack of m 2s in the exponent; $A(4,1) = 2^{2^{16}} - 3 = 2^{65536} - 3$, which exceeds even the number of atoms in the universe (which is 10^{80} according to current theories).

The definition translates very nicely into C++, but the task of expressing it in a nonrecursive form is truly troublesome.

5.8 Excessive Recursion

Logical simplicity and readability are used as an argument supporting the use of recursion. The price for using recursion is slowing down execution time and storing on the run-time stack more things than required in a nonrecursive approach. If recursion is too deep (for example, computing $5.6^{100,000}$), then we can run out of space on the stack and our program crashes. But usually, the number of recursive calls is much smaller than 100,000, so the danger of overflowing the stack may not be imminent.[2]

[2]Even if we try to compute the value of $5.6^{100,000}$ using an iterative algorithm, we are not completely free from a troublesome situation because the number is much too large to fit even a variable of double length. Thus, although the program would not crash, the computed value would be incorrect, which may be even more dangerous than a program crash.

However, if some recursive function repeats the computations for some parameters, the run time can be prohibitively long even for very simple cases.

Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

$$\text{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1) & \text{otherwise} \end{cases}$$

The definition states that if the first two numbers are 0 and 1, then any number in the sequence is the sum of its two predecessors. But these predecessors are in turn sums of their predecessors, and so on, to the beginning of the sequence. The sequence produced by the definition is

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

How can this definition be implemented in C++? It takes almost term-by-term translation to have a recursive version, which is

```
unsigned long Fib(unsigned long n) {
  if (n < 2)
    return n;
// else
    return Fib(n-2) + Fib(n-1);
}
```

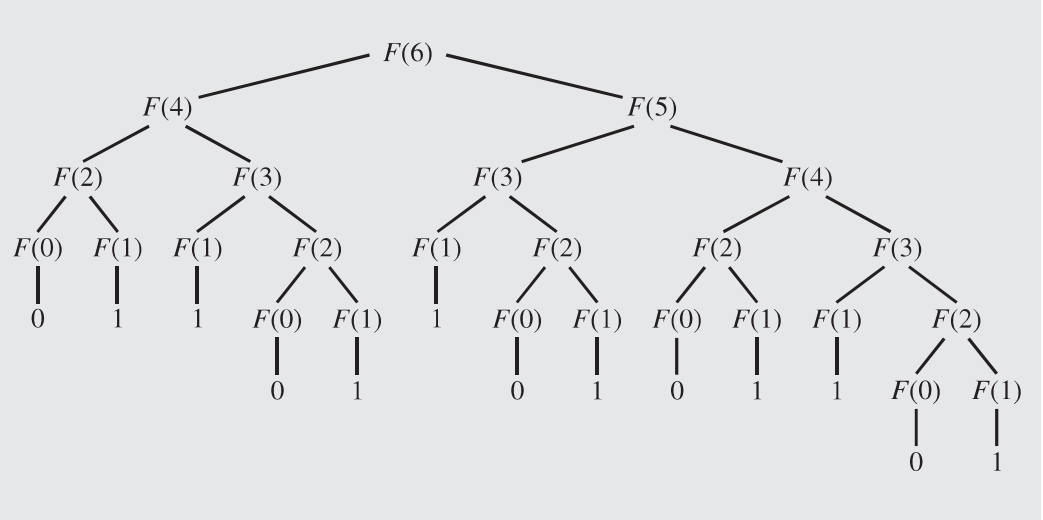
The function is simple and easy to understand but extremely inefficient. To see it, compute `Fib(6)`, the seventh number of the sequence, which is 8. Based on the definition, the computation runs as follows:

```
Fib(6) =                Fib(4)                  + Fib(5)
       =    Fib(2)        +        Fib(3)       + Fib(5)
       = Fib(0)+Fib(1)    +        Fib(3)       + Fib(5)
       =   0   +   1      +        Fib(3)       + Fib(5)
       =       1          + Fib(1)+   Fib(2)    + Fib(5)
       =       1          + Fib(1)+Fib(0)+Fib(1) + Fib(5)
```

etc.

This is just the beginning of our calculation process, and even here there are certain shortcuts. All these calculations can be expressed more concisely in the form of the tree shown in Figure 5.8. Tremendous inefficiency results because `Fib()` is called 25 times to determine the seventh element of the Fibonacci sequence. The source of this inefficiency is the repetition of the same calculations because the system forgets what has already been calculated. For example, `Fib()` is called eight times with parameter `n` = 1 to decide that 1 can be returned. For each number of the sequence, the function computes all its predecessors without taking into account that it suffices to do this only once. To find `Fib(6)` = 8, it computes `Fib(5)`, `Fib(4)`, `Fib(3)`, `Fib(2)`, `Fib(1)`, and `Fib(0)` first. To determine these values, `Fib(4)`, . . . , `Fib(0)` have to be computed to know the value of `Fib(5)`. Independently of this, the chain of computations `Fib(3)`, . . . , `Fib(0)` is executed to find `Fib(4)`.

FIGURE 5.8 The tree of calls for `Fib(6)`.



We can prove that the number of additions required to find `Fib(n)` using a recursive definition is equal to Fib$(n + 1) - 1$. Counting two calls per one addition plus the very first call means that `Fib()` is called $2 \cdot \text{Fib}(n + 1) - 1$ times to compute `Fib(n)`. This number can be exceedingly large for fairly small *n*s, as the table in Figure 5.9 indicates.

It takes almost a quarter of a million calls to find the twenty-sixth Fibonacci number, and nearly 3 million calls to determine the thirty-first! This is too heavy a price for the simplicity of the recursive algorithm. As the number of calls and the run time grow exponentially with *n,* the algorithm has to be abandoned except for very small numbers.

FIGURE 5.9 Number of addition operations and number of recursive calls to calculate Fibonacci numbers.

n	Fib(n+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1,973
20	10,946	10,945	21,891
25	121,393	121,392	242,785
30	1,346,269	1,346,268	2,692,537

An iterative algorithm may be produced rather easily as follows:

```
unsigned long iterativeFib(unsigned long n) {
    if (n < 2)
         return n;
    else {
         register long i = 2, tmp, current = 1, last = 0;
         for ( ; i <= n; ++i) {
             tmp = current;
             current += last;
             last = tmp;
         }
         return current;
    }
}
```

For each `n > 1`, the function loops `n - 1` times making three assignments per iteration and only one addition, disregarding the autoincrement of `i` (see Figure 5.10).

However, there is another, numerical method for computing Fib(n), using a formula discovered by Abraham de Moivre:

$$\text{Fib}(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

where $\phi = \frac{1}{2}(1+\sqrt{5})$ and $\hat{\phi} = 1 - \phi = \frac{1}{2}(1-\sqrt{5}) \approx -0.618034$. Because $-1 < \hat{\phi} < 0$, $\hat{\phi}^n$ becomes very small when n grows. Therefore, it can be omitted from the formula and

$$\text{Fib}(n) = \frac{\phi^n}{\sqrt{5}}$$

approximated to the nearest integer. This leads us to the third implementation for computing a Fibonacci number. To round the result to the nearest integer, we use the function `ceil` (for ceiling):

FIGURE 5.10 Comparison of iterative and recursive algorithms for calculating Fibonacci numbers.

		Assignments	
n	Number of Additions	Iterative Algorithm	Recursive Algorithm
6	5	15	25
10	9	27	177
15	14	42	1,973
20	19	57	21,891
25	24	72	242,785
30	29	87	2,692,537

```
unsigned long deMoivreFib(unsigned long n) {
    return ceil(exp(n*log(1.6180339897) - log(2.2360679775)) - .5);
}
```

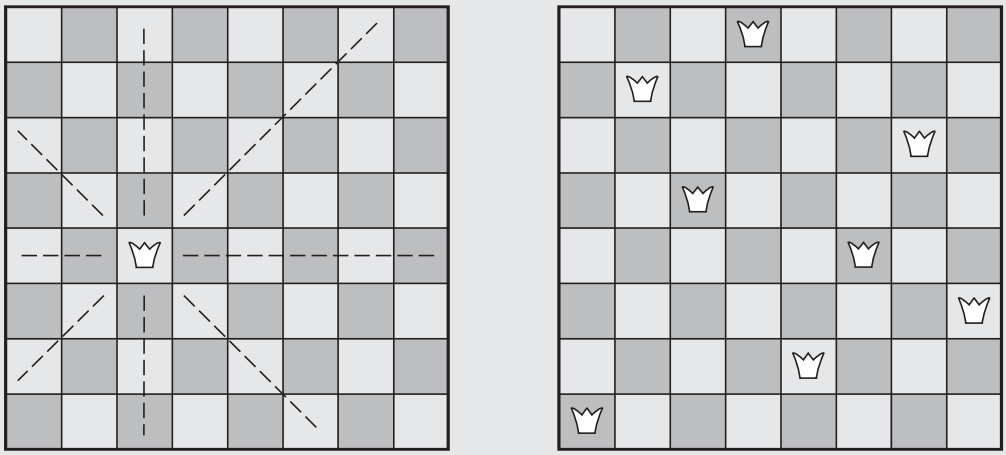
Try to justify this implementation using the definition of logarithm.

5.9 Backtracking

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called *backtracking,* and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

The eight queens problem attempts to place eight queens on a chessboard in such a way that no queen is attacking any other. The rules of chess say that a queen can take another piece if it lies on the same row, on the same column, or on the same diagonal as the queen (see Figure 5.11). To solve this problem, we try to put the first queen on the board, then the second so that it cannot take the first, then the third so that it is not in conflict with the two already placed, and so on, until all of the queens are placed. What happens if, for instance, the sixth queen cannot be placed in a nonconflicting position? We choose another position for the fifth queen and try again with the sixth. If this does not work, the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and then the

FIGURE 5.11 The eight queens problem.



(a) (b)

process restarts. This process requires a great deal of effort, most of which is spent backtracking to the first crossroads offering some untried avenues. In terms of code, however, the process is rather simple due to the power of recursion, which is a natural implementation of backtracking. Pseudocode for this backtracking algorithm is as follows (the last line pertains to backtracking):

```
putQueen(row)
    for every position col on the same row
      if position col is available
         place the next queen in position col;
         if (row < 8)
           putQueen(row+1);
         else success;
         remove the queen from position col;
```

This algorithm finds all possible solutions without regard to the fact that some of them are symmetrical.

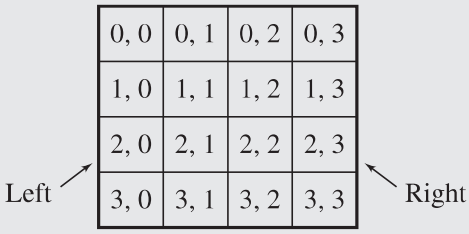
The most natural approach for implementing this algorithm is to declare an 8×8 array `board` of 1s and 0s representing a chessboard. The array is initialized to 1s, and each time a queen is put in a position (r, c), `board[r][c]` is set to 0. Also, a function must set to 0, as not available, all positions on row r, in column c, and on both diagonals that cross each other in position (r, c). When backtracking, the same positions (that is, positions on corresponding row, column, and diagonals) have to be set back to 1, as again available. Because we can expect hundreds of attempts to find available positions for queens, the setting and resetting process is the most time-consuming part of the implementation; for each queen, between 22 and 28 positions have to be set and then reset, 15 for row and column, and between 7 and 13 for diagonals.

In this approach, the board is viewed from the perspective of the player who sees the entire board along with all the pieces at the same time. However, if we focus solely on the queens, we can consider the chessboard from their perspective. For the queens, the board is not divided into squares, but into rows, columns, and diagonals. If a queen is placed on a single square, it resides not only on this square, but on the entire row, column, and diagonal, treating them as its own temporary property. A different data structure can be utilized to represent this.

To simplify the problem for the first solution, we use a 4×4 chessboard instead of the regular 8×8 board. Later, we can make the rather obvious changes in the program to accommodate a regular board.

Figure 5.12 contains the 4×4 chessboard. Notice that indexes in all fields in the indicated left diagonal all add up to two, $r + c = 2$; this number is associated with this diagonal. There are seven left diagonals, 0 through 6. Indexes in the fields of the indicated right diagonal all have the same difference, $r - c = -1$, and this number is unique among all right diagonals. Therefore, right diagonals are assigned numbers –3 through 3. The data structure used for all left diagonals is simply an array indexed by numbers 0 through 6. For right diagonals, it is also an array, but it cannot be indexed by negative numbers. Therefore, it is an array of seven cells, but to account for negative values obtained from the formula $r - c$, the same number is always added to it so as not to cross the bounds of this array.

FIGURE 5.12 A 4 × 4 chessboard.



An analogous array is also needed for columns, but not for rows, because a queen i is moved along row i and all queens $< i$ have already been placed in rows $< i$. Figure 5.13 contains the code to implement these arrays. The program is short due to recursion, which hides some of the goings-on from the user's sight.

FIGURE 5.13 Eight queens problem implementation.

```
class ChessBoard {
public:
    ChessBoard();    // 8 x 8 chessboard;
    ChessBoard(int); // n x n chessboard;
    void findSolutions();
private:
    const bool available;
    const int squares, norm;
    bool *column, *leftDiagonal, *rightDiagonal;
    int  *positionInRow, howMany;
    void putQueen(int);
    void printBoard(ostream&);
    void initializeBoard();
};

ChessBoard::ChessBoard() : available(true), squares(8), norm(squares-1)
{
    initializeBoard();
}
ChessBoard::ChessBoard(int n) : available(true), squares(n),
norm(squares-1) {
    initializeBoard();
}
```

FIGURE 5.13 (*continued*)

```
void ChessBoard::initializeBoard() {
    register int i;
    column = new bool[squares];
    positionInRow = new int[squares];
    leftDiagonal  = new bool[squares*2 - 1];
    rightDiagonal = new bool[squares*2 - 1];
    for (i = 0; i < squares; i++)
        positionInRow[i] = -1;
    for (i = 0; i < squares; i++)
        column[i] = available;
    for (i = 0; i < squares*2 - 1; i++)
        leftDiagonal[i] = rightDiagonal[i] = available;
    howMany = 0;
}
void ChessBoard::putQueen(int row) {
    for (int col = 0; col < squares; col++)
        if (column[col] == available &&
            leftDiagonal [row+col] == available &&
            rightDiagonal[row-col+norm] == available) {
            positionInRow[row] = col;
            column[col] = !available;
            leftDiagonal[row+col] = !available;
            rightDiagonal[row-col+norm] = !available;
            if (row < squares-1)
                 putQueen(row+1);
            else printBoard(cout);
            column[col] = available;
            leftDiagonal[row+col] = available;
            rightDiagonal[row-col+norm] = available;
        }
}
void ChessBoard::findSolutions() {
    putQueen(0);
    cout << howMany << " solutions found.\n";
}
```

Figures 5.14 through 5.17 document the steps taken by `putQueen()` to place four queens on the chessboard. Figure 5.14 contains the move number, queen number, and row and column number for each attempt to place a queen. Figure 5.15 contains the changes to the arrays `positionInRow`, `column`, `leftDiagonal`, and

FIGURE 5.14 Steps leading to the first successful configuration of four queens as found by the function `putQueen()`.

Move	Queen	row	col	
{1}	1	0	0	
{2}	2	1	2	failure
{3}	2	1	3	
{4}	3	2	1	failure
{5}	1	0	1	
{6}	2	1	3	
{7}	3	2	0	
{8}	4	3	2	

FIGURE 5.15 Changes in the four arrays used by function `putQueen()`.

positionInRow	column	leftDiagonal	rightDiagonal	row
(0 , 2 , ,)	(!a , a , !a , a)	(!a, a , a , !a , a , a , a)	(a, a , !a , !a , a , a , a)	0 , 1
{1}{2}	{1} {2}	{1} {2}	{2}{1}	{1}{2}
(0 , 3 , 1 ,)	(!a , !a , a , !a)	(!a, a , a , !a , !a , a , a)	(a, !a , a , !a , !a , a , a)	1 , 2
{1}{3}{4}	{1} {4} {3}	{1} {4}{3}	{3} {1} {4}	{3}{4}
(1 , 3 , 0 , 2)	(!a , !a , !a , !a)	(a, !a , !a , a , !a , !a , a)	(a, !a , !a , a , !a , !a , a)	0 , 1 , 2 , 3
{5} {6} {7} {8}	{7} {5} {8} {6}	{5} {7} {6} {8}	{6} {5} {8} {7}	{5}{6}{7}{8}

`rightDiagonal`. Figure 5.16 shows the changes to the run-time stack during the eight steps. All changes to the run-time stack are depicted by an activation record for each iteration of the `for` loop, which mostly lead to a new invocation of `putQueen()`. Each activation record stores a return address and the values of `row` and `col`. Figure 5.17 illustrates the changes to the chessboard. A detailed description of each step follows.

{1} We start by trying to put the first queen in the upper left corner (0, 0). Because it is the very first move, the condition in the `if` statement is met, and the queen is placed in this square. After the queen is placed, the column 0, the main right diagonal, and the leftmost diagonal are marked as unavailable. In Figure 5.15, {1} is put underneath cells reset to `!available` in this step.

FIGURE 5.16 Changes on the run-time stack for the first successful completion of `putQueen()`.

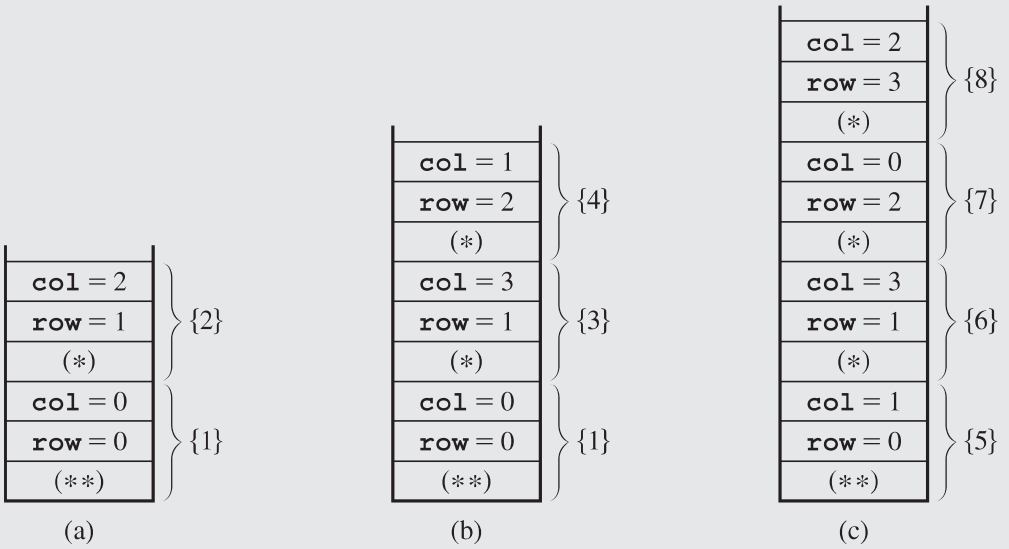
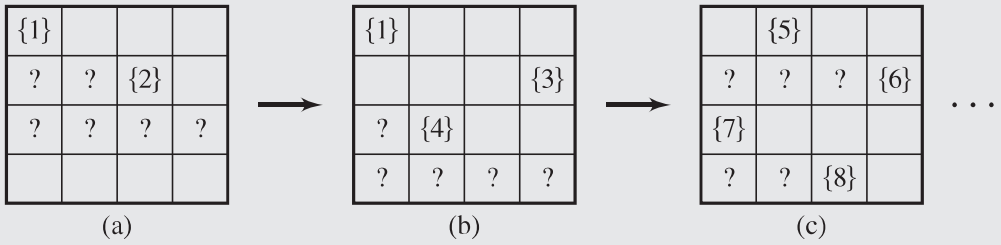


FIGURE 5.17 Changes to the chessboard leading to the first successful configuration.



{2} Since `row<3`, `putQueen()` calls itself with `row+1`, but before its execution, an activation record is created on the run-time stack (see Figure 5.16a). Now we check the availability of a field on the second row (i.e., `row==1`). For `col==0`, column 0 is guarded, for `col==1`, the main right diagonal is checked, and for `col==2`, all three parts of the `if` statement condition are true. Therefore, the second queen is placed in position (1, 2), and this fact is immediately reflected in the proper cells of all four arrays. Again, `row<3`. `putQueen()` is called trying to locate the third queen in row 2. After all the positions

in this row, 0 through 3, are tested, no available position is found, the `for` loop is exited without executing the body of the `if` statement, and this call to `putQueen()` is complete. But this call was executed by `putQueen()` dealing with the second row, to which control is now returned.

{3} Values of `col` and `row` are restored and the execution of the second call of `putQueen()` continues by resetting some fields in three arrays back to `available`, and since `col==2`, the `for` loop can continue iteration. The test in the `if` statement allows the second queen to be placed on the board, this time in position (1, 3).

{4} Afterward, `putQueen()` is called again with `row==2`, the third queen is put in (2, 1), and after the next call to `putQueen()`, an attempt to place the fourth queen is unsuccessful (see Figure 5.17b). No calls are made, the call from step {3} is resumed, and the third queen is once again moved, but no position can be found for it. At the same time, `col` becomes 3, and the `for` loop is finished.

{5} As a result, the first call of `putQueen()` resumes execution by placing the first queen in position (0, 1).

{6–8} This time execution continues smoothly and we obtain a complete solution.

Figure 5.18 contains a trace of all calls leading to the first successful placement of four queens on a 4 × 4 chessboard.

FIGURE 5.18 Trace of calls to `putQueen()` to place four queens.

```
putQueen(0)
  col = 0;
  putQueen(1)
    col = 0;
    col = 1;
    col = 2;
    putQueen(2)
      col = 0;
      col = 1;
      col = 2;
      col = 3;
    col = 3;
    putQueen(2)
      col = 0;
      col = 1;
      putQueen(3)
        col = 0;
        col = 1;
        col = 2;
        col = 3;
```

FIGURE 5.18 (*continued*)

```
        col = 2;
        col = 3;
    col = 1;
    putQueen(1)
      col = 0;
      col = 1;
      col = 2;
      col = 3;
      putQueen(2)
        col = 0;
        putQueen(3)
          col = 0;
          col = 1;
          col = 2;
          success
```

5.10 Concluding Remarks

After looking at all these examples (and one more to follow), what can be said about recursion as a programming tool? Like any other topic in data structures, it should be used with good judgment. There are no general rules for when to use it and when not to use it. Each particular problem decides. Recursion is usually less efficient than its iterative equivalent. But if a recursive program takes 100 milliseconds (ms) for execution, for example, and the iterative version only 10 ms, then although the latter is 10 times faster, the difference is hardly perceivable. If there is an advantage in the clarity, readability, and simplicity of the code, the difference in the execution time between these two versions can be disregarded. Recursion is often simpler than the iterative solution and more consistent with the logic of the original algorithm. The factorial and power functions are such examples, and we will see more interesting cases in chapters to follow.

Although every recursive procedure can be converted into an iterative version, the conversion is not always a trivial task. In particular, it may involve explicitly manipulating a stack. That is where the time–space trade-off comes into play: using iteration often necessitates the introduction of a new data structure to implement a stack, whereas recursion relieves the programmer of this task by handing it over to the system. One way or the other, if nontail recursion is involved, very often a stack has to be maintained by the programmer or by the system. But the programmer decides who carries the load.

Two situations can be presented in which a nonrecursive implementation is preferable even if recursion is a more natural solution. First, iteration should be used in the so-called real-time systems where an immediate response is vital for proper