Chapter 5
Introduction to Recursion

And often enough, our faith beforehand in a certain result is the only thing that makes the result come true.

— William James, *The Will To Believe*, 1897

Objectives

- To be able to define the concept of *recursion* as a programming strategy distinct from other forms of algorithmic decomposition.
- To recognize the paradigmatic form of a recursive function.
- To understand the internal implementation of recursive calls.
- To appreciate the importance of the *recursive leap of faith*.
- To understand the concept of *wrapper* functions in writing recursive programs.
- To be able to write and debug simple recursive functions at the level of those presented in this chapter.

Most algorithmic strategies used to solve programming problems have counterparts outside the domain of computing. When you perform a task repeatedly, you are using iteration. When you make a decision, you exercise conditional control. Because these operations are familiar, most people learn to use the control statements **`for`**, **`while`**, and **`if`** with relatively little trouble.

Before you can solve many sophisticated programming tasks, however, you will have to learn to use a powerful problem-solving strategy that has few direct counterparts in the real world. That strategy, called **recursion,** is defined as any solution technique in which large problems are solved by reducing them to smaller problems *of the same form*. The italicized phrase is crucial to the definition, which otherwise describes the basic strategy of stepwise refinement. Both strategies involve decomposition. What makes recursion special is that the subproblems in a recursive solution have the same form as the original problem.

If you are like most beginning programmers, the idea of breaking a problem down into subproblems of the same form does not make much sense when you first hear it. Unlike repetition or conditional testing, recursion is not a concept that comes up in day-to-day life. Because it is unfamiliar, learning how to use recursion can be difficult. To do so, you must develop the intuition necessary to make recursion seem as natural as all the other control structures. For most students of programming, reaching that level of understanding takes considerable time and practice. Even so, learning to use recursion is definitely worth the effort. As a problem-solving tool, recursion is so powerful that it at times seems almost magical. In addition, using recursion often makes it possible to write complex programs in simple and profoundly elegant ways.

5.1 A simple example of recursion

To gain a better sense of what recursion is, let's imagine you have been appointed as the funding coordinator for a large charitable organization that is long on volunteers and short on cash. Your job is to raise $1,000,000 in contributions so the organization can meet its expenses.

If you know someone who is willing to write a check for the entire $1,000,000, your job is easy. On the other hand, you may not be lucky enough to have friends who are generous millionaires. In that case, you must raise the $1,000,000 in smaller amounts. If the average contribution to your organization is $100, you might choose a different tack: call 10,000 friends and ask each of them for $100. But then again, you probably don't have 10,000 friends. So what can you do?

As is often the case when you are faced with a task that exceeds your own capacity, the answer lies in delegating part of the work to others. Your organization has a reasonable supply of volunteers. If you could find 10 dedicated supporters in different parts of the country and appoint them as regional coordinators, each of those 10 people could then take responsibility for raising $100,000.

Raising $100,000 is simpler than raising $1,000,000, but it hardly qualifies as easy. What should your regional coordinators do? If they adopt the same strategy, they will in turn delegate parts of the job. If they each recruit 10 fundraising volunteers, those people will only have to raise $10,000. The delegation process can continue until the volunteers are able to raise the money on their own; because the average contribution is $100, the volunteer fundraisers can probably raise $100 from a single donor, which eliminates the need for further delegation.

If you express this fundraising strategy in pseudocode, it has the following structure:

```
void CollectContributions(int n) {
    if (n <= 100) {
        Collect the money from a single donor.
    } else {
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
    }
}
```

The most important thing to notice about this pseudocode translation is that the line

Get each volunteer to collect **n/10** *dollars.*

is simply the original problem reproduced at a smaller scale. The basic character of the task—raise *n* dollars—remains exactly the same; the only difference is that *n* has a smaller value. Moreover, because the problem is the same, you can solve it by calling the original function. Thus, the preceding line of pseudocode would eventually be replaced with the following line:

```
CollectContributions(n / 10);
```

It's important to note that the **CollectContributions** function ends up calling itself if the contribution level is greater than $100. In the context of programming, having a function call itself is the defining characteristic of recursion.

The structure of the **CollectContributions** procedure is typical of recursive functions. In general, the body of a recursive function has the following form:

```
if (test for simple case) {
    Compute a simple solution without using recursion.
} else {
    Break the problem down into subproblems of the same form.
    Solve each of the subproblems by calling this function recursively.
    Reassemble the solutions to the subproblems into a solution for the whole.
}
```

This structure provides a template for writing recursive functions and is therefore called the **recursive paradigm.** You can apply this technique to programming problems as long as they meet the following conditions:

1. You must be able to identify **simple cases** for which the answer is easily determined.
2. You must be able to identify a **recursive decomposition** that allows you to break any complex instance of the problem into simpler problems of the same form.

The **CollectContributions** example illustrates the power of recursion. As in any recursive technique, the original problem is solved by breaking it down into smaller subproblems that differ from the original only in their scale. Here, the original problem is to raise $1,000,000. At the first level of decomposition, each subproblem is to raise $100,000. These problems are then subdivided in turn to create smaller problems until the problems are simple enough to be solved immediately without recourse to further subdivision. Because the solution depends on dividing hard problems into simpler ones, recursive solutions are often called **divide-and-conquer** strategies.