Algorithms

3. Procedures and Recursion

R K Shyamasundar

R K Shyamasundar is Professor of Computer Science at TIFR, Bombay and has done extensive research in various foundation areas of computer science.

In this article we introduce *procedural abstraction* and illustrate its uses. Further, we illustrate the notion of *recursion* which is one of the most useful features of procedural abstraction.

Procedures

Let us consider a variation of the problem of summing the first M natural numbers. The problem is: *Compute the partial sums of all numbers from 1 to M.*

That is, we have to compute

$$\Sigma_{i=1}^{j} \; i \qquad (1)$$

for all j such that $1 \leq j \leq M$. This means we have to compute M sums according to equation (1).

A Naive Solution. Let us recall the program (discussed in ***Resonance,*** Vol.1, No.3) for summing N numbers. The code segment for summing N numbers is given in *Table 1*. Using the segment "code", we can obtain the algorithm for the problem on

Table 1. Program segment "code"

```
count: = 0;
sum: = 0;
I: = 1;
while (count < N) do
sum: = sum + i;          (* sum contains the sum of first i numbers *)
i: = i + 1;              (* increment i to get the next number *)
count: = count + 1;      (* count keeps count of the numbers added *)
endwhile;                (* sum contains the sum of first N numbers *)
print sum;
```

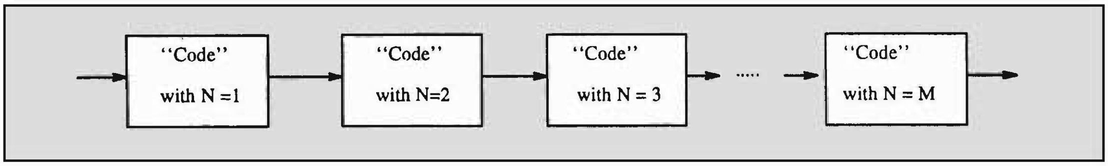

Figure 1 A naive solution for summing numbers.

hand as follows:

Place M copies of the program segment "code"(*Table 1*) in the form shown in *Figure 1*. Here, "code" with $N = j$ corresponds to the code shown in *Table 1* with N replaced by j.

In the program shown in *Figure 1,* we have repeated the algorithm M times and we can make the following observations. Each block is essentially a different instance of "code"; that is, the objects differ by the value to which N is initialized before the execution of the "code" block. Thus, we can now avoid the repetition of the "code" blocks by using an additional loop for setting the values of N. The program corresponding to such a refinement is shown in *Table 2.*

It uses an outer loop on the code segment "code" for passing the appropriate value for N. It is convenient to give the "code" a name so that it can be referred to (or used) later. In other words, the

Table 2. Program for finding all the partial sums

```
        N:=1;
while N ≤ M do
            count: = 0;
            sum: = 0;
            i: = 1:
          while (count <N) do
            sum: = sum + i;         (* sum contains the sum of first i numbers *)
            i: = i + 1;             (* increment i to get the next number *)
            count: = count + 1;     (* count keeps count of the numbers added *)
          endwhile:                 (* sum contains the sum of first N numbers *)
        print sum;
        N:=N+1;
endwhile
```

Table 3. Procedural abstraction for summing numbers

```
procedure SUMMING (N: integer);
    count: = 0;
    sum: = 0:
    i: = 1;
    while (count < N ) do
            sum: = sum + i;
            i: = i + 1;
            count: = count + 1;
    endwhile:
    print sum;
endprocedure
```

Procedural abstraction not only makes the program concise but also easily comprehensible; the latter aspect is very important for verifying the correctness (either formally or informally) of the program.

name given refers to the object obtained by abstracting away the differences among the different instantiations of the repeated code; this is referred to as *procedural abstraction* and the name given is referred to as the *procedure name*. To account for the differences among the distinct instantiations, we list the variables of the program (i.e., the aspects in which the several objects differ) along with the name of the procedure. The variables thus listed along with the procedure are referred to as *formal parameters*. These variables/parameters can further be divided into *input* and *output* variables; the *input* parameters are those through which you provide the input values to the program. These values form the basis on which the outputs are to be computed. In other words, procedures could be treated as relations between the input and the output parameters. Such an algorithmic segment is called a *procedure* (or subroutine) and the variables of the algorithmic segment are referred to as formal parameters. The procedural form is shown in *Table* 3. The body (i.e., the code between the keywords procedure and endprocedure of the procedure) is the same as the "code" given in *Table 1*.

The identifier "SUMMING" is the name given to the algorithmic segment and is also referred to as the name of the procedure. In the procedure SUMMING, *N* denotes a formal variable/parameter

Table 4. Computing N sums using procedure SUMMING

```
N: = 1;
while N < M do
    call SUMMING ( j ).
    N: = N+1
endwhile
```

which can differ from one invocation to another. The notation *N: integer* indicates that *N* can take any value from the domain of integers. In other words, *Table 3* defines procedure SUMMING. Having defined a procedure, we can use it as if it were another *basic command.* To clearly distinguish it from the basic commands, we use the keyword "call" to indicate its usage. For example, "call SUMMING(100)", corresponds to executing the above procedure with the initial input value of *N* equal to 100. Now using the procedural form, the program for computing (1) (that is, computing all the intermediate sums up to *M*) can be written as shown in *Table 4.*

Procedural abstraction leads to saving space for storing program code, easy comprehension and a good structure.

In *Table* 4, the command "call SUMMING (j)" denotes the execution of the procedure SUMMING with variable (i.e., the formal parameter) *N* taking the value of *j* for each call; in this command *j* is referred to as the *actual parameter* as it is this value that is used in the execution of the procedure SUMMING. The keyword "call" denotes the invocation of the procedural segment. This keyword is omitted in several representations since it can be understood implicitly. It can be easily seen that the algorithms shown in *Tables 2* and 4 are concise. Assuming one has understood "code", it can be said that the program shown in *Table 4* is more comprehensible than the one shown in *Table 2.* Thus, procedural abstraction not only makes the program concise but also easily comprehensible; the latter aspect is very important for verifying the correctness (either formally or informally) of the program. To summarize, procedural abstraction is based on the two principles indicated on the next page:

- Concentrate on the properties shared by several objects by abstracting away the differences.
- A succinct parametrization of the differences.

Procedural abstraction leads to saving space for storing program code, easy comprehension and a good structure. In fact, it becomes very handy when the same code-segment can be invoked for different purposes; these aspects as well as the power of parameters will become clear in the next article when we discuss a simple logo like programming language.

One of the usual techniques of problem solving is to break the problem into smaller problems. From the solution of these smaller problems one obtains a solution for the original problem.

Recursion

One of the usual techniques of problem solving is to break the problem into smaller problems. From the solution of these smaller problems, one obtains a solution for the original problem. Consider the procedural abstraction described above. It is possible to visualize the given procedure as being decomposed into a set of procedures. It may so happen that a smaller procedure (i.e., the sub-problem) is also of the same form as the original procedure, except that in '*measure*' it is '*smaller*' than the original. Assuming that we know the solution of problems for a certain finite set of *base cases,* we can then obtain a clear solution for the original problem or the procedure. A procedural abstraction which refers to itself is called *a recursive procedure.* We illustrate this powerful concept through the following example.

Example 1 (Towers of Hanoi) : This example is based on an ancient puzzle originating in a monastery in Tibet. We are given three

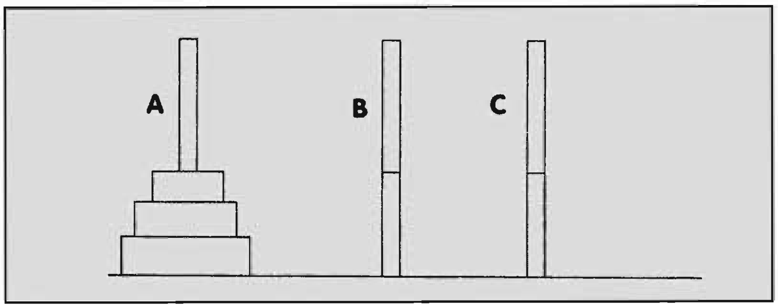

Figure 2 Towers of Hanoi problem.

rods and n disks of different sizes. The disks can be stacked on the rods, thereby forming 'towers'. Let the N disks initially be placed on rod A in the order of decreasing size as shown in *Figure 2.* The task is to move the N disks from rod A to rod C such that they are ordered in the original way. This has to be achieved under the following constraints:

- In each step only the topmost disk can be moved from one rod and placed on top of the disks on another rod.
- A disk may never be placed on top of a smaller disk.
- Rod B may be used as an auxiliary store.

The problem is to find an algorithm which performs this task. First let us consider the solution for $N = 2$. The solution is trivial : shift the smaller disk on A to rod B; then shift the larger disk on A to rod C; now shift the smaller disk on B to rod C (on top of the larger disk). Let us abstract it by the procedure *move_rod* $(N_0 A, B, C)$ where N_0 (equal to 2 in this case) is the number of disks on A; B and C do not have any disks initially; and finally all the disks are transferred to C and they are in proper order. Let us see how we can arrive at the solution for other values of N using mathematical induction.

A procedural abstraction which refers to itself is called a *recursive procedure.*

Now we have a solution for the base case: *move_rod* (N, A, B, C) is certainly possible for $N = 2$ (base case). For inductively arriving at the successive steps, we have to derive the solution for $N_0 + 1$ from the solution ofN_0.

Steps for $N_0 + 1$: The computation of*move_rod* $(\mathrm{N}_0 + 1, A, B, C)$ can be derived from *move_disk* $(N_0 + 1,\mathrm{A}, B, C)$ by the following steps:

- *move_rod* (N_0, A, C, B);
  N_0 disks are moved from A to B using C as auxiliary rod.
- *move_disk* (A, C);
  $(N_0 + 1)$th disk is moved from A to C directly.
- *move_rod* (N_0,B, A,C)
  The N_0 disks which are in proper order are transferred from B to C using A as an auxiliary rod.

Table 5. Procedure for the Towers of Hanoi problem

```
Steps:
procedure move_rod (N:Nat_Number A:name, B:name, C:name);
  if N > 1 then
      move_rod (N - 1, A, C, B);
      move_disk (A, C)
      move_rod (N - 1, B, A, C)
  else move_disk (A, C)
endprocedure
```

It may be noted that *move_disk* can be treated as a basic command.

The Towers of Hanoi problem is based on an ancient puzzle originating in a monastery in Tibet.

What is special in these steps? We have split the original problem into problems of smaller size. Further, the solution of the smaller problem is obtained by invoking the same procedure with appro– priate input. That is, *the main procedure calls itself.* This aspect of the procedures where one uses the ability of a procedure to *call itself* is referred to as *recursion*. This is one of the very important features of programming. The algorithm is shown in *Table 5.*

Trace of the steps for $N = 4$

- Step 1:
  move_rod (3, *A*, *B*, *C*)
- Step 2 : The above call leads to ($N > 1$ holds):
  move_rod (2, *A*, *C*, *B*)
  move_disk (*A*, *C*)
  move_rod (2, *B*, *A*, *C*)
- Step 3 : The call to *move_rod* (2, *A*, *C*, *B*) leads to (N> 1 holds) :
  move rod (*1*, *A*, *B*, *C*)
  move disk (*A*, *B*)

```
    move_rod (1, C, A, B)
    move_disk (A, C)
    move_rod (2, B, A, C)
```

- Step 4 : The call to *move_rod* (1, *A*, *C*, *B*) leads to (N=1 holds):

```
    move_disk (A, C)
    move_disk (A, B)
    move_rod (1, C, A, B)
    move_disk (A, C)
    move_rod (2, B, A, C)
```

- Step 5 : Taking the basic actions on the first two calls to *move_disk* leads to:
  {The smallest disk has been moved to peg C (corresponding to *move_disk* (*A*, *C*,)};
  {The next smallest disk (as the first has already been moved) has been moved to peg B (corresponding to *move_disk (A, B)*)}

```
    move_rod (1, C, A, B)
    move_disk (A, C)
    move_rod (2, B, A, C)
```

At the end of step 5 the smallest disk is in peg C, the next larger in B and the largest disk remains at peg A.

- Step 6 : The call to *move rod* (1, *C*, *A*, *B)* (N=1 holds) leads to : {The smallest disk has been moved to peg C and the next smallest disk is on peg B and hence, the largest (3rd) is on peg A};

```
    move_disk (C, B)
    move_disk (A, C)
    move_rod (2, B, A, C)
```

- Step 7 : Executing the basic action *(move_disk)* leads to:
  {The smallest disk has been moved to peg C and the next smallest disk is on peg B and hence, the largest (3rd) is on peg A};

{The smallest disk on peg C is moved to peg B on top of the disk larger than it (corresponding to *move_disk (C, B))* };
{The largest disk on peg A is moved to peg C (which is empty now) corresponding to *move_disk (A, C)* };

move_rod (2, *B*, *A*, *C*)

- Step 8 : With the largest disk on peg C and the other disks on peg B (in the appropriate order), a call to *move_rod* (2, B, A, C) leads to:

  move_rod (1, *B*, *C*, *A*)
  move_disk (*B*, *C*)
  move_rod (1, *A*, *B*, *C*)

- Step 9 : With the largest disk on peg C and the other disks on peg B (in the appropriate order), (and hence, peg A is empty) the three basic actions can be rewritten as follows:
  {move the disk on B (the smallest) to peg A (corresponding to *move_disk (B, A))*};
  {move the disk from B (the second largest) on to the top of C already containing the largest disk (corresponding to *move_disk (B, C)*)};
  {move the smallest on peg A onto the top of peg C already containing the other two disks in appropriate order (corresponding to the call *move_disk (A, C))*};

At the end of step 7 peg A is empty. Peg C has the largest disk and peg B the two disks in the right order.

Now, we have realized the objective of transferring the disks from peg A to C as per the protocol. Further, the program terminates as there are no calls left.

Now let us see what happens when the number of disks is less than 3. The case when $N = 1$ is trivial, as the disk is transferred straight from peg A to peg C. Now, let us consider the case $N = 2$. From the procedure, it can be seen that the call *move_rod* (2, *A*,*B*,*C*) reduces to

move_rod (1, *A*, *C*, *B*);
move_disk(*A*, *C*);
move_rod (1, *B*, *A*, *C*);

This further rewrites into:

move_disk (A, B);
move_disk (A, C);
move_disk (B, C);

The basic actions lead to:

move the smallest disk from A to B (corresponds to *move_disk (A, B));*
move the largest disk from A to C (corresponds to *move_disk(A, C));*
move the smallest disk from peg B to peg C on top of the largest disk (corresponds to *move_disk (B,C)).*

Example 2. Recursive program for gcd

Let us see from the analysis done earlier, whether we can arrive at a recursive program for computing the gcd of two positive numbers. From the earlier discussion, we have:

$gcd(m,n) = gcd\ (m \text{ rem } n, n).$

A recursive algorithm to compute gcd is very elegant.

Now, we can derive a recursive program from the following observations :

- To simplify, let us replace m rem n by $m - n$; this step should be convincing since division could be treated as repeated subtraction.

Table 6. A recursive gcd program

```
procedure gcd (m:integer, n:integer);
   if m=n then gcd is n
   else if m > n then gcd(m-n,n)
        else gcd(m, n-m)
        endif
   endif
endprocedure
```

- In the algorithm discussed in the second article (***Resonance***, Vol.1, No.3, 1996), the gcd algorithm terminates when the remainder becomes zero. Since we are using subtraction, it can be easily seen that the condition can be replaced by $m = n$.
- We should subtract the smaller number from the larger number. Thus, the roles of m and n may have to be reversed. Fortunately, $gcd(m,n) = gcd(n,m)$.

The program is shown in *Table 6*.

Discussion

In the previous sections, we have illustrated the advantages of procedural abstraction and introduced recursive procedures. The trace of the various invocations of the procedure calls, for the Towers of Hanoi example, shows how the procedures are invoked with new parameters. In a sense, one can consider the code yet to be executed as a *push-down stack* of procedure calls to be executed; in a push-down stack, you can access only the topmost element and hence we will be executing a procedure which entered the stack last (more about such data-structures will become clear in the forthcoming articles). Thus, one can assume that the program has terminated once the stack is empty and the last procedure has terminated. It can be observed that the recursive program for gcd looks simple and easy to understand. However, from this observation, we should not conclude that whenever possible one should use a recursive program. These aspects will become clear from the subsequent articles in this series.

Address for correspondence
R K Shyamasundar
Computer Science Group
Tata Institute of
Fundamental Research,
Homi Bhabha Road
Mumbai 400 005, India.

Suggested Reading

E W Dijkstra. A Short Introduction to the Art of Programming. CSI Publication. 1977.

R G Dromey. How to Solve it by Computer. Prentice Hall International. 1982.

D Harel. Algorithmics: The Spirit of Computing. Addison-Wesley Publishing Co. Inc. 1987.
This is one ofthe most lucidly written books on the topic.