Algorithms

2. The While-Construct

R K Shyamasundar

In this article, we consolidate the introductory concepts developed in the previous article of the series (*Resonance*, Vol.1, No.1) and develop the *iterative construct*, one of the most important control constructs used to describe algorithms.

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

In the last article, we learnt about *assignment* and other basic commands which are imperative commands to a processor. Further, we discussed the basic control structures which include sequential composition and the *test* (or more specifically, the *if-then-else*) construct. Using these constructs, we developed the basic flowchart language for describing algorithms. In this article, we continue the discussion of control constructs and their representation using flowcharts. We describe the 'while-construct' which is one of the most widely used iterative control constructs for describing algorithms.

Iteration

We concluded the last article with the question: "Is it possible to obtain a concise flowchart to find the sum of the first N natural numbers?" We hinted that it was possible, by using a construct in which the number of times a set of commands is executed depends on the values of certain variables. Such a construct, referred to as the 'while-construct', is shown in *Figure 1*. The construct is interpreted as follows: Test for B; if the test leads to the answer "NO", then we have reached the end; otherwise the control goes to block S, after which the process repeats.

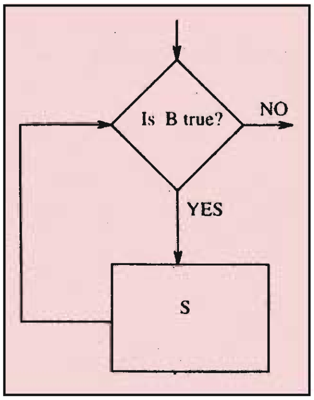

***Figure 1** An important algorithmic construct called 'while loop'.*

It is important to note that B is false (usually denoted by $\neg$B where $\neg$ denotes the logical negation unary operator) on termination of the

In the while-construct the number of times a set of commands is executed depends on the values of certain variables.

while-loop. The textual representation of the construct is:

while B *do* S *endwhile*

When these operations terminate, we can assert that $\neg$B (i.e., complement of B) holds.

Example 1: Going back to summing the first N natural numbers.

We describe an algorithm that can be used for any *N*. The idea is to keep track of the numbers we have already added and exit when we have added all the *N* numbers. The flowchart shown in *Figure 2* describes such an algorithm. We see that the same algorithm works for any value of *N* (fixed a priori). The textual algorithm (referred to as 'code') corresponding to the flowchart is given in *Table 1*. This algorithm solves the problem of adding the first *N* natural numbers for any value of *N*. We may add the box *'read N'*, shown in *Figure 3*, to the top of the flowchart given in *Figure 2*. It accomplishes the task of substituting the value of *N* in the flowchart of the given program. In other words, when *read N* is executed, the variable *N* takes the value from the given input.

Euclid's Algorithm

We learn to compute the greatest common divisor (*gcd*) in our primary school arithmetic classes. Although popularly known as Euclid's Algorithm, it was described by Euclid's predecessor Eudoxous. The ancient Chinese had also discovered this algorithm.

Example 2: Euclid's Algorithm.

We now describe Euclid's algorithm for computing the greatest common divisor *(gcd)* of two positive integers *m* and *n*. By *gcd*, we mean the largest positive number that exactly divides both *m* and *n*. A naive way of obtaining the *gcd* is to factor both the numbers and take the common factors. However, such a scheme is quite tedious.

The Greek philosopher Euclid provided a better solution to this problem. Let us see the reasoning behind Euclid's algorithm. Let *x* be equal to *gcd* (m,n) where $m \geq n$. Then, we observe the following:

- $x \leq n$ since $n \leq m$. That is, the maximum value of x is bounded by the smaller of the two numbers (i.e., by n).
- $x = n$ implies that n exactly divides m.
- From the definition of *gcd*, we see that *gcd* (m,n) = *gcd* (n,m).

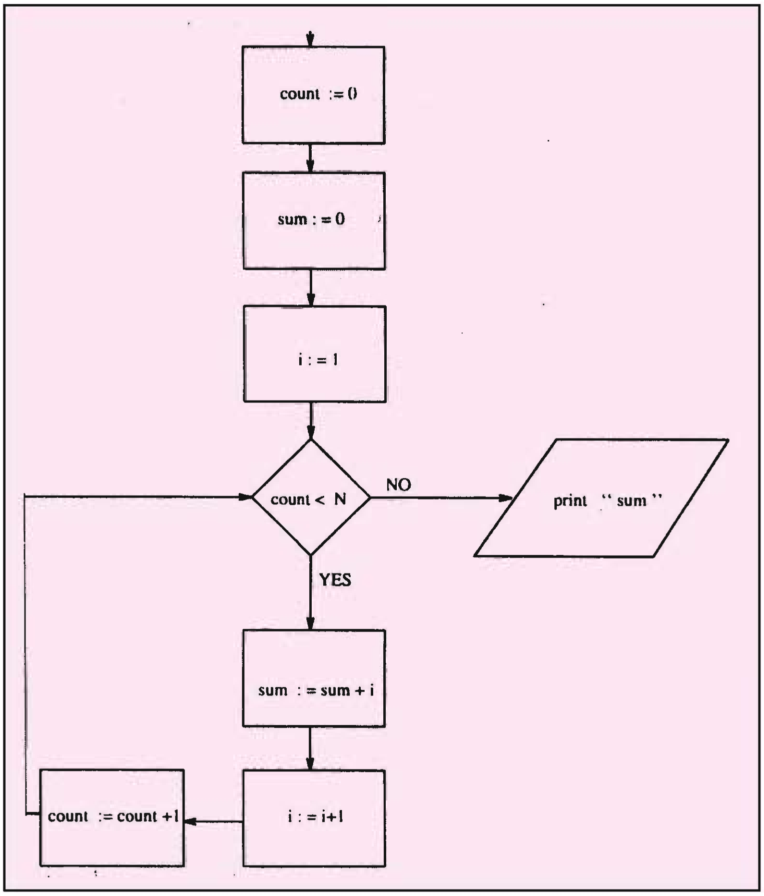

Figure 2 A flowchart to sum the first N natural numbers (N to be read separately).

Figure 3 Box 'read N' to be composed with the flow-chart of Figure 2.

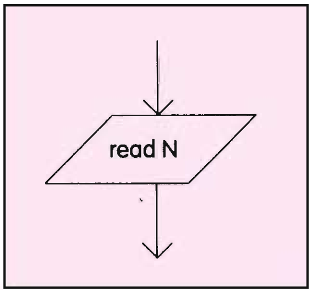

Table 1. Textual representation of the flowchart

```
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 counts the numbers added * )
endwhile;        (* sum contains the sum of first N numbers *)
print sum ;
```

- Let us suppose that n does not exactly divide m. Then, we have $m = p \times n + r$ for some p and $0 < r < n$. We can conclude that x must divide n and r. This follows since x must divide both the numbers m and n. Can we say anything stronger? Yes, we can say that *gcd* (m,n) is the same as *gcd* (n,r) (follows from the definition of *gcd*). The same argument can be applied for r and n. Note that the bound on the candidate for x gets reduced each time; now x is bounded by r. This is a crucial step in ensuring that the algorithm terminates.

Assuming m is greater than or equal to n, the flowchart for computing the *gcd* is shown in *Figure 4*. The operator rem used

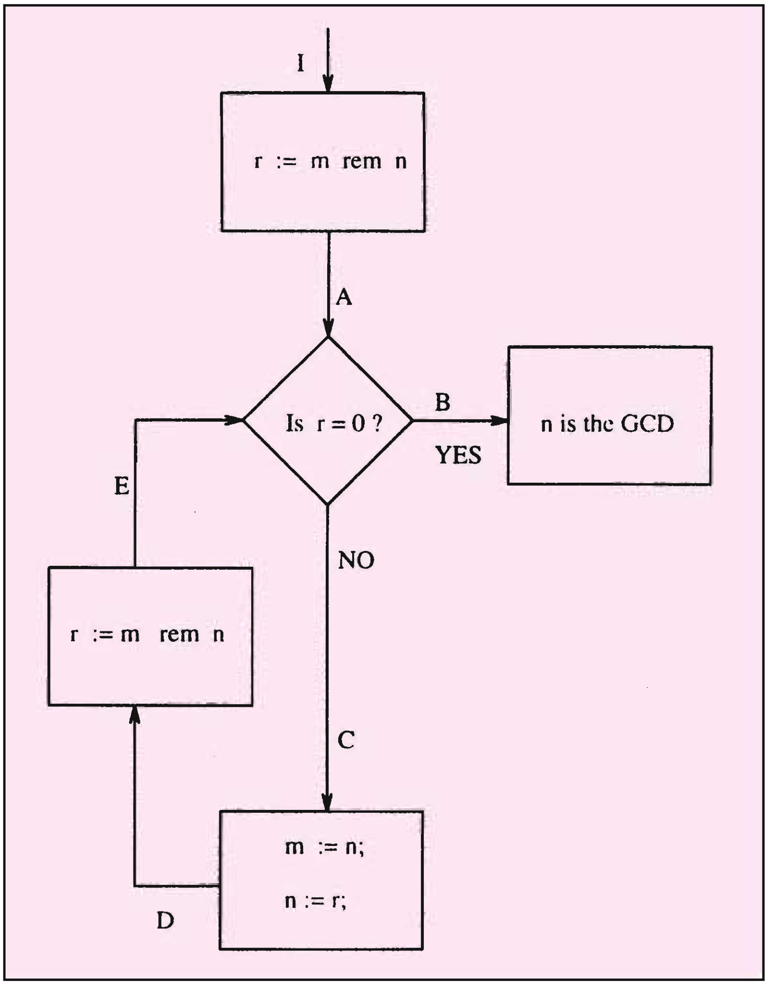

Figure 4 Flowchart for computing* gcd (m,n) *using Euclid's algorithm.

The greatest common divisor (gcd) of two positive integers m and n is the largest positive number that divides both m and n.

Table 2. Trace of gcd (8,6)

Step	Action
1.	$r := 8$ rem $6 = 2$
2.	$r \neq 0 \longrightarrow m := 6;\ n := 2;\ r := 6$ rem $2 = 0$
3.	$r = 0 \longrightarrow$ '2' is the gcd

here is defined by: p rem q = remainder obtained on dividing p by q. The trace of Euclid's Algorithm for $m = 8$ and $n = 6$ is shown in *Table 2*.

Now let us see how we can informally argue that the algorithm indeed computes what we want. For convenience, we have labelled the arrows in the flowchart. By observing the flow of information, we can assert the following facts at the labels:

- At label A: r is set to the remainder obtained on dividing m by n. Hence, $0 \leq r < n$; m and n remain unchanged (i.e., $m = p \times n + r$ assuming $m \geq n$).
- At C: the remainder r is not equal to zero.
- At D: m is set to n and n is set to the remainder. Also, we have $m > n$. Can we say that the *gcd* of the original m and n and the new m and n are the same? From the discussion given above, we can indeed assert this statement.
- At B: The remainder r is equal to zero — leading to the *gcd*.

Example 3: Computing a factorial.

The familiar definition of *factorial* is

$$fact\,(n) = n! = 1 \times 2 \times \ldots \times n, \quad n \geq 0.$$

How do we derive an algorithm for computing *fact*? We first express *fact* (i) in terms of the *fact* (j) for $j < i$. Note that

$$fact\,(1) = 1 \qquad (1)$$
$$fact\,(i) = 1 \times \ldots \times i \qquad (2)$$
$$fact\,(i+1) = 1 \times \ldots \times i \times (i+1) \qquad (3)$$

The algorithm for computing the factorial (fact) uses the recurrence relation that fact $(i+1)$ equals fact $(i) \times (i+1)$.

Table 3. Algorithm for Computing a Factorial

```
fact : = 1 ;
i : = 0 ;
while i ≤ N do
        i := i+1 ;
        fact := fact * i
end while
print fact;
```

Assuming *fact* (0) = 1, and combining (2) and (3) we get the following relations (recurrence):

$$fact\,(0) = 1 \qquad (4)$$
$$fact\,(i+1) = fact\,(i) \times (i+1) \qquad (5)$$

The reader may observe that various interesting programs can be developed along the same lines for computing the 'sine' function from its series, the 'Fibonacci numbers', etc.

Now, we can get a simple algorithm using the relations (4) and (5). In the algorithm, we start with an initialization of *fact* (0) to be 1. The successive factorials can then be obtained by multiplying the immediate preceding factorial (computed in the previous step) by the next natural number. The algorithm is described in *Table 3.*

Example 4: Finding the 'integer' square root.

We devise an algorithm to find the approximate (integer) square root of a fixed number $n \geq 0$. For example, the integer square root of 4 is 2, integer square root of 5 is 2, and integer square root of 10 is 3. That is we have to find an a such that

$$a^2 \leq n < (a+1)^2 \qquad (6)$$

The basic scheme would be to start from a good guess and move on to the next guess if the number chosen does not satisfy the required property. It is important that when we move from the current guess to the next guess we do not miss the actual number we are looking for. Thus, starting with 0 as the first guess and incrementing it by one every time till (6) is satisfied, will eventually yield the result. But it will be too 'expensive'. We can learn

Table 4. Finding the Integer Square Root.

```
(* Finding the integer square root of $n$ *)
$a := 0$ ;                        (* lowest guess *)
$b := n+1$ ;                      (* largest guess *)
while $(a+1) \neq b$ do           (* get the average guess *)
    $d := (a+b) \div 2$ ;         (* $\div$ denotes integer division*)
    if $(d * d) \leq n$ then
        $a := d$                  (* refined lower guess *)
    else $b := d$                 (* refined largest guess *)
    endif
endwhile
```

To find the integer square root the basic scheme would be to start from a good guess and move on to the next guess if the number chosen does not satisfy the required property. It is important that when we move from the current guess to the next guess we do not miss the actual number we are looking for.

something from the relation (6) itself. We simultaneously guess a *lower* bound (say l) and an *upper* bound (say u) and update these two bounds appropriately. At the initial stage, 0 is a candidate for l and $n+1$ is a candidate for u. Next, how do we update l and u? By taking the square root of the numbers involved in the relation (6) we can derive the following relation

$$a \leq \sqrt{n} < (a+1) \tag{7}$$

Thus, a is bounded above by $\sqrt{n}$. Let us try to reduce the interval (l, u) by half, by setting l or u to $(l + u)/2$ such that the condition $l < u$ is still satisfied. The reader can check that this strategy will not skip the number we are looking for. Note that l will never reach the upper bound. This idea has been used to develop the algorithm described in *Table 4*.

Example 5: Finding an 'approximate' square root.

In the previous section, we developed an algorithm for finding the integer square root of a number. The integer square root can be considered as a crude approximation to the square root of a number. Let us see whether we can modify the above technique and compute the square root of any positive number such that it differs from the actual square root by at most some given *tolerance*

The important question in iteration is: " When do we stop? " We stop when the iterates stop decreasing, i.e. when there are no more representable values with the given machine accuracy.

limit. Since square roots of natural numbers need not be natural numbers, such a modification will permit us to find the square root of decimal numbers also. It may be pointed out that in general, we cannot compute the exact value of the square root of a number, as the number may not be representable in the given machine accuracy.

Now, we will adapt the above algorithm (given in *Table 4*) and compute the approximate square root of a number. Let us assume that $x_0 > 0$ is the first guess of the square root of the given number a; a is assumed to be a positive non-zero number. Then, a/x_0 is also an approximation to $\sqrt{a}$ and

$$\begin{aligned} a/x_0 &< \sqrt{a} \quad \text{if } x_0 > \sqrt{a} \\ a/x_0 &> \sqrt{a} \quad \text{if } x_0 < \sqrt{a}. \end{aligned}$$

The interesting fact is that the average of x_0 and a/x_0, say x_1, is also an approximate square root and satisfies the property

$$x_1 \geq \sqrt{a}$$

The equality holds only if $x_0 = \sqrt{a}$. Note that it is not necessary that x_0 be greater than or equal to $\sqrt{a}$. We can repeat the process of obtaining the next approximate square root; then, the $(i+1)^{th}$ approximation (denoted by x_{i+1}) is given by

$$x_{i+1} = (x_i + a/x_i)/2$$

The fact that the new approximate square root is better than the earlier one follows from:

$$x_1 \geq x_2 \geq \ldots \geq x_{i+1} \geq \sqrt{a}$$

From this relation, we infer that the value gets refined through the process of finding the next approximation from the current one. The successive approximates of x_i are referred to as the *iterates.* Do note that each iterate is better than the earlier ones. The important question is: "When do we stop?" We stop when the iterates stop decreasing; in the above case, they stop decreasing when there are no more representable values between $\sqrt{a}$ and x_i with the given machine accuracy. Suppose *error* represents the accuracy to which

Table 5. Finding the Approximate Square Root

```
(* Finding the approximate square root of A *)
a := A;                              (* A is the given number *)
E := error;                          (* error is the given accuracy *)
xold := $X_0$;                       (* initial guess *)
xnew := (xold + a / xold) / 2 ;      (* refined root *)
while (xnew - xold) > E do
      xold := xnew;
      xnew := (xold + a / xold) / 2 ; (* refined root *)
endwhile
```

the number can be represented in the given computer. Then, we can stop whenever $(x_{i+1} - x_i)$is less than or equal to this quantity. Assuming that we have been given an *initial guess* and an *error* which can be tolerated, the program for finding the approximate root is given in *Table 5*.

Note the following:

- The division operator '/' used in *Table 5* denotes the usual division operation and not the integer division operation used in *Table 4*.
- Unless the initial guess is the correct guess, the equality in the relation among the iterates does not hold. Thus, if we start with an incorrect guess even for a natural number having an exact square root, we will not get the exact root using this method.
- The number of iterations before the program terminates depends on the starting values (initial guesses); it is of interest to note that there are procedures to arrive at these initial guesses for the technique discussed above.

The method described above for computing the approximate square root is referred to as Newton's method for finding $\sqrt{a}$ after the famous English mathematician Isaac Newton.

In *Table 5*, we have essentially solved the nonlinear equation

Iterative Method

In an iterative method, we compute a new approximate solution in terms of the previous one. The new approximation should be *better* than the old one. Iterative methods are sometimes called trial and error methods. This is because each successive iterate relies on the degree by which it differs from the previous one. For this method to be of value, it is necessary to show that the refined solutions eventually become more accurate. Further, one should define a condition for stopping the iterations as in most cases the iterate will never reach the correct answer. However, finding such conditions is difficult.

The reason for referring to the while-construct defined in the earlier sections is also based on these observations.

The method for computing the approximate square root is referred to as Newton's method for finding $\sqrt{a}$, after the famous English mathematician Isaac Newton.

$x^2 = a$. The method can be extended to find the n^{th} root of the equation $x^n = m$ and it is usually referred to as the Newton-Raphson method.

Discussion

In the previous sections and the previous article, we have learnt several constructs such as: assignment and basic commands, sequential composition, *if-then-else*, and the *while* construct. We can categorize these constructs into two classes:

- *Imperative Commands*: These are instructions to the processor. Constructs such as assignment and other basic commands belong to this class.
- *Control Commands*: These are commands which reflect the way in which the various instructions are sequenced. The *if-then-else* statement provides conditional sequencing of instructions, and the *while*-construct provides conditional sequencing based repeatedly on a given condition. These constructs are referred to as *control structures. The* control structures abstract the way the commands can be executed on a machine. Such an abstraction is often referred to as *control abstraction*. It must be evident to the reader that one can devise various other control structures. For instance, one can devise a construct where the control enters the statement block first and is tested at the end of the statement block execution. This is different from the while-construct where a condition is tested before entering a statement block. One such construct is the *repeat-until* construct. For example, *repeat* S *until* B *endrepeat* can be interpreted as: Repeatedly execute S until the condition B holds. Thus, when the statement terminates, we can conclude that B holds.

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

Having looked at the above basic constructs, it is natural to ask the following questions:

- Are the above mentioned constructs general for all programming purposes and if yes, in what sense?

- In the description of algorithms and programming languages, what is the role of control abstraction?
- What are the inherent limitations of the algorithmic processes?

In future articles in this series, we will show that these constructs are powerful and can be used to encode any algorithm. In the next article, we will discuss procedural abstraction and one very widely used programming technique called recursion in the context of procedural abstraction. We will also provide a relative comparison of the iterative and the recursive constructs in the description of algorithms.

We can categorize the constructs studied so far into two classes: *imperative commands,* which are instructions to the processor and *control commands,* which reflect the way various instructions are sequenced.

Suggested Reading

D E Knuth. Art of Computer Programming. Volume 1. Addison-Wesley Publishing Co. 1972.

E W Dijkstra. A Short Introduction to the Art of Programming. Computer Society of India. 1977.

G Polya. How to Solve It. Princeton University Press. 1973.

R G Dromey. How to Solve it by Computer. Prentice-Hall of India, New Delhi. 1990.

It is a pleasure to acknowledge the critical constructive comments and suggestions from the series editor.

The ultimate folly... "The worst thing that can happen — will happen (in the 1980s) — is not energy depletion, economic collapse, limited nuclear war, or conquest by a totalitarian government. As terrible as these catastrophes would be for us, they can be repaired within a few generations. The one process ongoing in the 1980s that will take millions of years to correct is the loss of genetic and species diversity by the destruction of natural habitats. This is the folly our descendants are least likely to forgive us.". *(E O Wilson, Harvard Magazine, January-February 1980).*

The discovery of the Mobius strip ... In 1858, a scientific society in Paris offered a prize for the best essay on a mathematical subject. In the course of coming up with an essay for this competition, August Ferdinand Mobius, a mathematician in Leipzig, Germany, 'discovered' the surface that now bears his name. It is called the Mobius-strip.