AN ALGORITHM TO DESIGN LOOPS

Amruth N. Kumar
Computer Science, T.A.S.
Ramapo College of New Jersey
505 Ramapo Valley Road
Mahwah, NJ 07430-1680
Phone: (201) 529 7712
(amruth@ramapo.edu)

ABSTRACT

Learning to write correct loops is hard. Often, CS 1 students get overwhelmed by the host of concerns that can significantly affect the correctness of a loop.

In this paper, we develop an algorithm to methodically guide students through designing and writing loops. The algorithm systematizes the concerns that must be borne in mind when designing loops. It draws the attention of students to the pitfalls as well as possibilities while designing loops. The algorithm can be used to design either a counter-controlled or a logic-controlled loop, and is independent of the language in which the loop is coded.

Instructors can use the algorithm to teach not only design but also analysis of loops. When used to analyze a loop, the algorithm reinforces the requirements of a loop, and inspires students to voluntarily seek out missing details in their loops.

INTRODUCTION

CS 1 students find loops hard to comprehend, because loops have a temporal behavior that is not evident from looking at the code. Teaching them semantics of a loop, using either a flowchart or a locus-of-control superscribed on the loop helps them analyze loops. However, this does not prepare them to design their own loops for a problem statement. This problem is especially pronounced among non-Computer Science majors or students with weak math backgrounds who take CS 1, as in our case, where the course counts towards general education credits.

Particularly vexing about designing a loop are the numerous issues that must be taken into account, which significantly affect the correctness and completeness of the loop. In this paper, we systematize these issues in the framework of an algorithm to design loops. The steps in the algorithm help a student methodically put together a loop, while also drawing the student's attention to the issues that must be addressed to verify
correctness of the loop. The algorithm can be used by students to design a loop for a problem statement, as well as analyze the correctness of a given loop.

The algorithm we propose is based on the existence of the following four components in a loop: condition, action (the iterated tasks), initialization and updating of variables. It is discussed after presenting the following classification of loops in class:

- **Counter-controlled loops**: used when the number of iterations is known. The loop is terminated when a counter variable reaches its final value. The template for the syntax of the loop in C is:

  ```c
  for( initialization; condition; updating )
  {
    action
  }
  ```

- **Logic-controlled loops**: used when the number of iterations is not known. The loop is terminated when the value of a condition variable reaches a sentinel value. These loops can again be categorized into:

- **Pre-test loops**: used when the loop may not be iterated at all. The condition is checked before the action is carried out in the loop. The template for the syntax of the loop in C is:

  ```c
  initialization
  while( condition )
  {
    action  // may be interchanged with updating
    updating
  }
  ```

- **Post-test Loop**: used when the loop is iterated at least once. The condition is checked after the action is carried out in the loop. The template for the syntax of the loop in C is:

  ```c
  do{
    initialization/updating
    action  // may be interchanged with updating
  } while( condition );
  ```

The algorithm may be used to design any of the above loops. It is independent of the language in which a loop is coded. In this paper, we have used the algorithm to develop an example each of a logic-controlled and a counter-controlled loop, both in pseudocode form.

In the past, proof-theoretic approaches have been used to derive and evaluate loops, wherein, pre-conditions, post-conditions and loop-invariants are expressed as axioms [1]. Students who are not mathematically inclined are reluctant to follow these approaches based on axiomatic semantics. What we present is an algorithmic alternative to these approaches, which is based on operational semantics. It has been our
experience that students are more willing to try the algorithm, and are more successful using it.

In this paper, first, we outline the algorithm and use it to design a logic-controlled loop. Next, we list the additional steps necessary to design a counter-controlled loop.

THE ALGORITHM

Consider the following problem, which is a candidate for a logic-controlled loop. We will use it to illustrate the algorithm:

"Read in characters until the first non-alphabetic character is encountered. For every character read, print the character four removed in the alphabet, as the cipher character corresponding to it. (e.g., A becomes an E, B an F, and so on.) Finally, print the number of alphabetic characters processed."

The algorithm to design loops is as follows:

**Step 1: Design the condition.**
Since the condition decides whether a loop is iterated or terminated, this is the most important component of a loop. In the above example, the condition is "until the first non-alphabetic character is encountered." If charvar is the name of a character variable, the condition may be written as:

```c
while alphabetic(charvar) // condition
```

**Step 2: Identify the loop variables, including those referenced in the condition.**
Loop variables are all the variables that are referenced before being assigned in a loop. These include variables referenced in the condition of a loop. In the above example, `charvar` is a loop variable.

**Step 3: Initialize the loop variables.**
Since loop variables are referenced in a loop, it must be ensured that they have a valid value before the loop is entered. In the above example, `charvar` must be initialized. Therefore, we have:

```c
read charvar // initialization
```

```c
while alphabetic(charvar) // condition
```

**Step 4: Design the action in the loop.**
Action includes the tasks that must be done in an iterative fashion in the loop. Designing action consists of an important sub-step:

- **Determine what must be iterated, and what may not.**
Novices often have the problem of distinguishing between what needs to be done only once (either before or after a loop), and what needs to be done iteratively, within a loop.

In the above example, the encryption of `charvar` must be printed on every iteration, and hence, must be part of action of the loop. The number of alphabetic characters processed must be printed only once, at the termination of the loop, and hence, must appear as a statement after the loop. However, novices often include the
statement to print the character count within the loop, generating unnecessary print-outs.

In order to help them determine what must be iterated, and what may not, we offer the following guidelines for reading a problem statement:

- Sentences in a problem statement which include the following phrases refer to tasks that must be iterated, i.e., parts of action of a loop:
  - “for each”, “for all” and “for every”
  - “do continuously” (infinite loop)
  - “for a series of values” or “for a sequence of values”.
  - “do n times”, or “for n values”
  - “do from p to q”, or “for values of x = l,m,n,p,q”. In the latter case, the values l, m, n, p and q must be in a series (arithmetic, geometric, etc.)

- Sentences in a problem statement which include the following phrases refer to tasks that must be carried out after a loop:
  - “Finally”, “In the end”, or “On termination”
  - “total value”, “final value”, “the result of”
  - conclusive words such as “Finished”, “Done”, etc.

More importantly, the above guidelines may be used by instructors to write clear problem statements which elicit correct responses from students, especially in tests. The guidelines help instructors narrow down the reasons for incorrect test answers to incomprehension or incorrect comprehension of the topic of loops.

In the above example, for every character read, its encrypted equivalent must be printed. Hence, we have:

```
read charvar       // initialization
while alphabetic(charvar) // condition
    print cipher(charvar)      // action
```

(Note that we use indentation to set off nested blocks, such as the loop body.)

After the loop terminates, the number of characters processed must be printed. We need another loop variable to count the number of characters processed, say count. It must be initialized. Therefore, we have:

```
count = 0          // initialization
read charvar       // initialization
while alphabetic(charvar) // condition
    print cipher(charvar)      // action
print count         // post-termination
```

**Step 5: Update the loop variables in the loop.**

Updating occurs within a loop, whereby fresh values are assigned to loop variables. The primary objective of updating is to nudge the values of loop variables referenced in loop condition, in a direction that ensures that the loop will eventually terminate.

In the above example, updating includes reading a new character into charvar, and incrementing count since another character has been processed. We now have:
count = 0 // initialization
read charvar // initialization
while alphabetic(charvar) // condition
    print cipher(charvar) // action
    read charvar // updating
    count = count + 1 // updating
print count // post-termination

Step 6: Verify relative order of update and action.
Loop variables may be updated either before or after action. We will analyze three possible cases.

If updating occurs before action, initialized values of loop variables cannot be used in action on the first iteration. Therefore, if a loop variable is referenced in action, it must be updated after the action. (Exceptions will be discussed later.)

Often, novices switch the order of updating and action. This results in bugs that they find hard to detect. In the above algorithm, for instance, if charvar is updated before the encrypted character is printed, the loop will still work, although, it will not print out encryption of the first character, and will print out encryption of the first non-alphabetic character! This behavior does not intuitively lead a debugging novice to suspect that the order of updating and action has been reversed. Hence, drawing their attention to the above possibility by means of an explicit step in the algorithm is helpful.

The statement updating count can be placed either before or after action in the loop. This is possible because count is not referenced in the action. If a loop variable is not referenced in the action of a loop, it can be updated anywhere in the loop.

Consider the following extension to the above problem statement:
“For every character read, print its ordinality (1, 2, 3, and so on) in addition to the cipher character corresponding to it.”

Now, the variable count must be referenced in action. It was initialized to 0, to account for the case when the loop is not iterated at all. This value is one removed from the value count should have on the first iteration of action. Therefore, count must be updated before action:

count = 0 // initialization
read charvar // initialization
while alphabetic(charvar) // condition
    count = count + 1 // updating
    print count // action
    print cipher(charvar) // action
    read charvar // updating
print count // post-termination

If a loop variable is referenced in action, but has been initialized to an irrelevant value, or to a value which is one update away from the value it must have on the first iteration of action, the loop variable is updated before action.
In all the above cases, however, the following requirement must be met: A loop variable must be updated only after any other variables that are needed to update it are themselves initialized/updated. E.g., if \( count = count + stepvar \) is how \( count \) is updated, \( count \) must be updated only after \( stepvar \) is initialized/updated.

**Step 7:** Verify where the loop variables referenced in action are updated, vis-a-vis the values to which they are initialized.

If a loop variable is initialized to the value it must have on the first iteration of action, it is updated after the action in which it is referenced. Otherwise, it is updated before the action.

In the above example, \( count \) is initialized to 0, which is one update away from the value it must have on the first iteration of action. Therefore, \( count \) is updated before action. If it is not required to print out the number of characters processed in the loop, we could have initialized \( count \) to 1 instead, and updated it after action. The former design is preferable if we need to keep a count of the number of iterations (or some other measure related to it), whereas, the latter design is more readable.

**Step 8:** Finally, desk check the loop.

Choose a few sequences of values to be assigned to loop variables which are part of loop condition. For each sequence of values, simulate the loop, step by step, iteration by iteration, to verify that the loop works as designed.

This step helps detect bugs in a loop. Especially with loops, a few minutes of manual desk checking can save a lot more time later, on using a debugger to step through the loop. Desk checking promotes an intuitive understanding of a loop, that is not only hard to get using a mechanical debugger, but also very useful when using such a debugger.

In order to effectively test a loop using desk checking, it is important to carefully choose sequences of values for its loop variables referenced in its condition. Following are some categories of sequences that must be used to test a loop:

- Normal sequence - which results in at least one iteration of a loop, and terminates the loop;
- Null sequence - which results in no iteration of a loop (not applicable to post-test loops);
- Infinite sequence - which does not terminate a loop. This sequence is especially useful to test loops which process finite data structures;
- Pathological sequence - where values within a sequence are not random. Examples include:
  - All the values in a sequence are the same (e.g., enqueue is the only operation ever chosen for a queue);
  - Values in a sequence are monotonic (e.g., input list to a sorting loop is already sorted);
  - The domain of values can be sub-divided, and all the values in a sequence belong to only one/some of the sub-domains (e.g., all the cards dealt in a card game are spades);
In general, the non-normal sequences correspond to extremities in the range of values assignable to length, variability of values, order of values, etc. in sequences. When more than one variable is referenced in the condition of a loop, various combinations of sequences may also have to be used to test the loop.

**COUNTER-CONTROLLED LOOPS**

A counter-controlled loop is much simpler than a logic-controlled loop: it uses a counter variable in its condition. Initialization, update and termination values for the counter variable are collectively called loop parameters. In most languages, counter-controlled loop constructs provide syntactic slots for initialization, condition and update of the counter variable on the very first line of a loop code.

Consider the following problem, which is a candidate for a counter-controlled loop. We will use it to illustrate extensions to the algorithm for counter-controlled loops:

"Read in the size of a class. Read in the grades of all the students in the class. Calculate the average grade in the class."

**Step 1:** The condition in a counter-controlled loop is simpler than that in a logic-controlled loop: a counter variable is compared with a termination value. Since grades must be read for all the students in the class, the termination value is the `class_size`. Therefore, the condition may be written as:

```
for counter <= class_size    // condition
```

**Step 2:** Loop variables include the counter variable(s), loop parameters - if they are in the form of variables (as opposed to constants), and any other variables referenced in loop action.

**Step 3:** The counter variable must be initialized. Loop parameters, if in the form of variables, must also be initialized. In the above example, `counter` and `class_size` are two loop variables that must be initialized. Hence, hewing to the usual syntax of counter-controlled loop constructs, we have:

```
read class_size    // initialization
for( counter = 1    // initialization
    counter <= class_size )    // condition
```

**Step 4:** A new grade must be read on every iteration, and hence, must be part of action of the loop. By definition, average must be calculated only once, after the loop, based on the sum and number of grades. This requires the use of another loop variable: an accumulator variable to calculate the running sum of the grades, say, `sum`. (This variable is used in this loop, primarily for efficiency. Alternatively, we could write another loop just to add the grades, as suggested by top-down decomposition of the
problem where, adding the grades together is a separate step from reading in the
grades.) sum must be initialized. Hence, we have:

read class_size       // initialization
sum = 0               // initialization
for( counter = 1      // initialization
     counter <= class_size ) // condition
     read grade      // action
average = sum/class_size // post-termination

Note that grade is not a loop variable by our definition because it is initialized before
referenced in the loop.

The relationship between loop action and loop variables used in the condition is
often a source of confusion for novices, especially in counter-controlled loops. Novices
are often under the misconception that a loop variable referenced in the condition must
also be referenced in loop action, and vice versa. Due to this misconception, they often
attempt to write loops such as:

for( loopvar = 1, loopvar <= class_size, loopvar = loopvar + 1 )    read loopvar

Disabusing them of this misconception helps free them to be more generous with using
new loop variables, and not mistake frugal use of variables for either necessity or
efficiency, leave alone correctness.

The possible relationships between loop action and loop variables that appear in
loop condition may be categorized as follows:

- Participatory: Loop variables used in the condition are also referenced in action.
Examples include loop variables referenced in assignment, expressions, function
calls, print statements and array subscripts.
- Non-participatory: Loop variables used in the condition are not referenced in
action. Only their termination value is of relevance to action. They merely act as
counters and sequencers of iterations of action.

The above categorization applies to logic as well as counter-controlled loops. The
categorization helps clarify under what circumstances loop variables could be referenced
in action, and when programmers should not feel obliged to contrive to use them in
action.

Step 5: The counter variable must be updated. Many languages disallow updating of
loop parameters. Even when a language permits updating of loop parameters, it should
be avoided to minimize the risk of creating an infinite loop, not to mention an
unreadable loop.

In the above example, counter and sum must be updated. Again, using the usual
syntax of counter-controlled loop constructs, we have:

read class_size       // initialization
sum = 0               // initialization
for( counter = 1      // initialization
     counter <= class_size ) // condition
     read grade      // action
average = sum/class_size // post-termination
counter <= class size  // condition
counter = counter + 1  // updating
read grade            // action
sum = sum + grade     // updating
average = sum/class size  // post-termination

Step 6: Since grade is used to update sum, sum is updated after action. For counter-controlled loops, most programming languages specify the order between updating of counter variable and action. Hence, the rules we developed in the previous section for logic-controlled loops (where the programmer has control over placement of updating), are not applicable to the counter variable. In the above example, we assume C specification [2], where updating is done after action.

Step 7: Since most programming languages specify that counter variable is updated after action, counter variable is initialized to the value it must have on the first iteration of action. Similarly, since sum is updated after action, it is initialized to the value it must have on the first iteration, until the first grade is read, viz., 0, the identity element of addition.

Step 8: The sequence of values is more clearly defined for the counter variable used in the condition of a counter-controlled loop in terms of loop parameters. Therefore, more specific tests can be applied to a counter-controlled loop during desk checking:

- To test for termination, verify that update value has the same sign as the expression (termination value - initialization value). The update value must not be the identity element for the operation used in updating, such as 0 for addition, or 1 for multiplication.
- Verify that the loop iterates correct number of times. The number of iterations of a loop is given by [3] max( 0, floor( (t - i + u) / u ) ) where i is the initialization value, t the termination value and u the update value. Conversely, if a loop must iterate n times over consecutive values starting at an initialization value p, (e.g., “Print the squares of 9 numbers starting with 13.”), the termination value should be n + p - 1, and not n + p as most novices would incorrectly conclude.
- Sequences with the following characteristics should be used to desk check a counter-controlled loop:
  - termination value < initialization value (Null sequence if update value is positive, Normal sequence otherwise)
  - termination value = initialization value (results in 1 iteration)
  - termination value > initialization value (Null sequence if update value is negative, Normal sequence otherwise)
DISCUSSION

So far, we have used the algorithm to teach logic-controlled loops in CS 1. Our CS 1 is part of the general education curriculum - even non-majors take the course to fulfill general education science requirement. Therefore, there is a wide range in the ability and math aptitude of our students. In such a setting, we find the above algorithm to write loops especially helpful to students.

The algorithm prevents the paralysis of action that strikes many novice students, who would otherwise not know where to begin when designing a loop for a problem statement. After introducing the algorithm, we observed our students explicitly (audibly) follow its steps while writing loops on the blackboard for the class. They were also much quicker to point out errors in loops written on the blackboard once they were introduced to the algorithm. We believe, the algorithm helps students analyze a problem more systematically, and evaluate their own solution in greater detail.

Whereas formal methods of loop correctness [1], popular in our curriculum today, help students analyze loops, our algorithm also helps students synthesize loops.

Whereas formal methods expect a certain level of math understanding from our students, our algorithm does not. Our experience also bears out that by using the algorithm to teach loops, instructors can cover the topic in shorter time, yet in greater detail because it streamlines discussion of all the relevant issues.

REFERENCES

