How to Write Loops

Writing perfect code the first time is something of a "Holy Grail" among programmers. By that I mean that most programmers long to do it, but the vast majority consider its attainment to be the stuff of legend.

Writing loops is one area where programming errors often crop up. Several years ago, however, I happened upon a technique developed by **Doug Cooper**, the Berkeley professor of "*Oh! Pascal*" fame, for building loops. I found that this technique **really does** increase your chances of building correct loops the first time, and it's worth the effort it takes to learn it.

Goal, Bounds & Plan

The first step in building a successful loop is to be able to describe (and separate) the loop's **bound** from the loop's **goal**, and then come up with a **plan** for reaching your goal.

The **bound** is the portion that makes it work "mechanically", while the **goal** of the loop is the work that you want to accomplish. The **plan** is the strategy you'll follow to both reach the bounds and, if possible, meet your goal.

Here's an example problem that we can use to examine the difference:

> *How many characters are in a sentence? Count the characters in a string until a period is encountered. If the string contains any characters, then it will contain a period. Count the period as well.*

Using this problem statement, you'll find that

- the **goal** of the loop is to **count the characters** which precede a period.
- the **bounds** of the loop are "a period was encountered."
- the **plan** is to a) "read" a character, and b) increment a counter

We can use the **same** bounds with a **different** goal:

> *Print each character in a string until a period is encountered. Then, add an exclamation point and print a newline.*

Notice that the **bound is the same**, but the **goal is different**. We're not counting, now we're printing. Our plan would change as well. Instead of adding one to a counter, in step b), we'd simply print the character, and, **after** the loop is finished, we'd print again.

Implementing the Plan

How do you put your plan into action? What tactics do you use to implement your strategy? Start by looking at the loop **topology**.

Examine this **guarded** loop. You can see that there are

- actions that occur **before** the loop is encountered
- a **test** that determines whether the **loop is entered**
- actions that occur **inside** the loop body
- actions that occur **after** the loop is complete

These four "faces" of a loop are called the **precondition**, the **bound**, the **action** or **operation** and the **postcondition**.

Remember these terms; we'll use them to determine exactly **where** to start working on our loop.

The Loop Bounds

If you've ever bought a new appliance (like a chain-saw, maybe), the first thing you need to learn is **how to turn it off**. That's where you should **always start writing your loops**; ask yourself: **What will make this loop quit?**

Making sure your loops quit at the right point is the **single most important** thing that you can do. Let's take our example with the loop bounds written in **pseudocode**.

```
// Step 1: Establish the loop bounds
// Before the loop
while letter is not a period
{
   // inside the loop
}
// After the loop
```

The Bounds Precondition

The bounds make sure that you exit the loop at the correct time. Next you have to that make it possible to **enter** the loop. Step asks **"How do I get out?"**, while Step 2 asks: "**How do I get into the loop?**" Look at the bounds condition:

```
while letter is not a period
```

1. **What** is letter?
2. **Where** did it come from?
3. **How** did it get a value that I can check?

If you were to write the bounds in C++, using your editor, your code would not compile because it refers to variables which don't yet exist. **Bounds precondition** statements **create the variables** used in the test, and **initialize** each to some meaningful state.

```
// Step 2: The bounds precondition
str <- string supplied to the problem
pos <- 0
letter <- str.at(pos)
while letter is not a period
{
   // Inside the loop
}
```

In our example, `str` is the string we've been given. We need two variables, `pos` which is the position (or index) of the character we want to examine, and `letter`, initialized with the first character in `str`.

If `letter` does not contain the first character in `str`, then you have no assurance that you will **ever enter the loop—** the value of `letter` will be unknown. (In C++ it will be some random value.)

Advancing the Loop

Now it's time to advance the loop, which means adding statements to the loop body that **move closer to the bounds** on each repetition. Let's see why. If you leave the body empty, what will happen?

1. If `str` begins with a period the loop will not be entered. The program works and reports that there are no characters before the period.
2. Otherwise, when the loop in entered, **nothing** in the body changes the value of `letter`, so there is **no way out** of the loop; it repeats over and over, **endlessly**. **Endless (infinite) loops** are common errors. Your IDE will appear as if it were "hung".
3. To avoid endless loops, be sure the statements inside the loop body **change something** tested in the loop bounds. Here, just store the next character, like this:

```
str <- string supplied to the problem
pos <- 0
letter <- str.at(pos)
while letter is not a period
{
   // Step 3: Advance the loop
   pos <- pos + 1
   letter <- str.at(pos)
}
```

At this point, the mechanical portion of your loop—the part that makes it "work", so to speak—is finished. You should be able to compile your code (once you've "translated it" into C++, of course), and it should run correctly.

The Loop Goal

The whole purpose of a loop is to get some real work done, to accomplish the **goal**. Up until now, we've ignored the goal portion entirely, because we want to make sure the loop **works**, before we put it **to work**.

The code to carry out the goal of the loop is written in a **different** order than that for the loop mechanics:

- start in the **precondition** area
- move to the **operations** in the loop body
- then deal with the **postcondition**

The Goal Precondition

The purpose of every loop is to **produce information**. Many loops **count** things and **add** things, for which you create **counters** and**accumulators**. For this step, **create and initialize the variables** necessary to **carry out the goal** of the loop.

Ask yourself: "**What information does this loop produce?**" Then, **create and initialize variables** to store that information. In our case, we now need a `counter` to store that information.

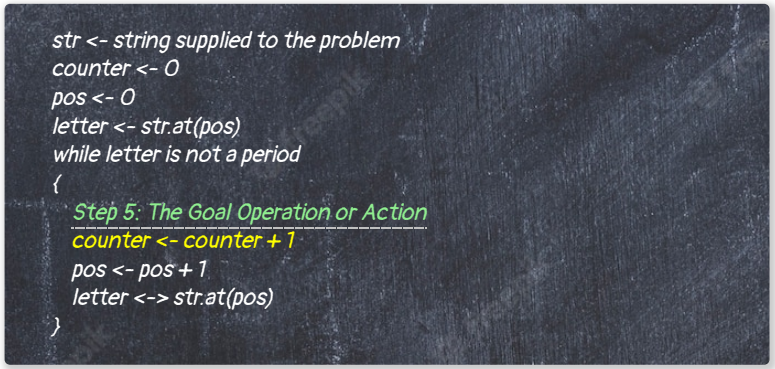
```
// Step 4: The goal preconditions
str <- string supplied to the problem
counter <- 0
pos <- 0
letter <- str.at(pos)
while letter is not a period
{
   pos <- pos + 1
   letter <-> str.at(pos)
}
```

Loop Operations

A loop can have many different goals, so it is hard to generalize about this step. A good start is to look at the variables you created in the last step and ask: "**How are these variables processed** to produce the desired output?"

Are you calculating a **sum**, **counting** different occurrences, or **transforming** existing values into new values? The statements inside the loop body will almost always **update the variables created in the goal precondition** in some way.

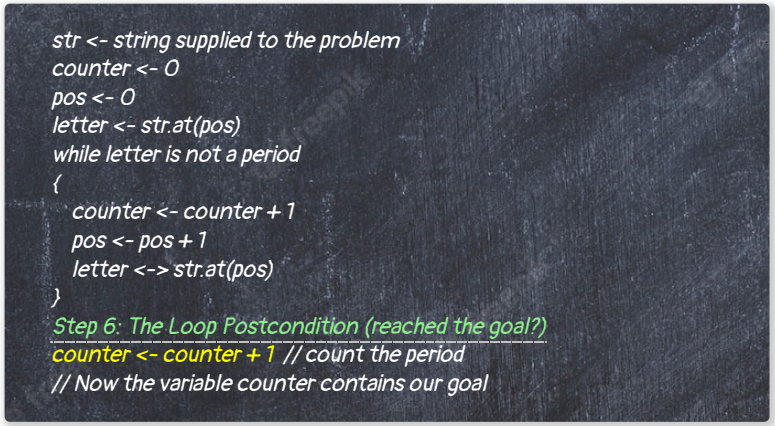
In our case, we need to increment our counter every time through the loop, since each time we read a character (that isn't a period), we want to count it.



The Postcondition

Loops end when reaching their bounds. not when they've reached the goal!
After the loop is over, you have to ask yourself: "**Has my loop reached its goal?**"

In our case, the problem asked us to count the period as well, and we haven't done that. So, after the loop is over we'll have to add a **postcondition** statement like this:



Now our loop has correctly counted the number of characters in the first sentence. **Or has it?** Next we'll look at two small details that I glossed over.

Loop Guards

The problem statement said that the input would have a period if characters were entered. But, what should we do if the input string is empty? We should skip the loop altogether.

You can modify the loop to handle this case by **guarding** the loop with an ***if*** statement.

```
Guarding the loop for an empty string
counter <- -1
If str is not empty then
{
   counter <- 0
   . . .
}
If counter is -1 then the goal is not reached
Otherwise counter contains the goal
```

The `counter` is set to `-1` **before** the loop guard. After the loop guard, if it still `-1` it means that the loop was **never entered**; you'll have to handle that in the **post-condition** processing.

Necessary & Intentional Bounds

If we change the problem so that it doesn't include the guarantee that every input string that contains characters will have a period, things are more complicated.

Here is what you need to ask yourself: "**Can my loop reach its bounds?**"

If there **is no period** in `str` then **obviously it cannot**. It will continue consuming any memory that appears after the string, **or**, it will crash. Neither are desirable.

A secondary condition designed for such eventualities is called a **necessary bound**. When we run out of characters, we must stop, even if our **intentional bound** is not reached. Here's our code modified to handle this complication.

```
// Adding a necessary bounds
counter <- -1
len <- str.size()
If  str != ""
{
   counter <- 0
   pos <- 0
   letter <- str.at(0)
   While pos < len and letter is not a period
   {
     ...
   }
   if letter is a '.' then counter <- counter + 1
   else counter <- -2
}
If counter is -1 the string was empty
Else if counter is -2 there was no period
Else counter contains the goal
```

The post-condition **now handles three cases**: a string with a period, the empty string, and a string with no period.

Sentinel Loop Patterns

How do you handle problems where the loop reads data from the user until some special value, or **sentinel**, is found to signal the end of the input? This is called a **sentinel loop** and its logical structure is:

```
Read a value
If the value is equal to the sentinel then
   Exit the loop
Process the value
```

There is no easy test at the beginning of the loop; you don't know when the sentinel is encountered **until you've read a value**. There are three ways to **rearrange** the statements to handle this situation.

The Primed Loop Pattern

The **primed-loop pattern** is named after the old-west water pump which required users to pour water down the well to establish suction, before pumping began.

Here's what that looks like in pseudocode:

```
Prompt user and read in a value
While the value is not the sentinel
  Process the data value
  Prompt user and read next value
```

This is the **classic way** to process sentinel data. The code used to read each data value is **duplicated, before the loop and at the end of the loop**.

Here's a **primed-sentinel-loop** that sums a sequence, using `0` as the sentinel:

```
cout << "Add integers. Enter 0 when done." << endl;

int total = 0, value = -1;
cout << "> ";
cin >> value;               // Read before the loop
while (value != 0)          // Check for the sentinel
{
    total += value;         // No sentinel? Process
    cout << "> ";           // Prompt and read next item
    cin >> value;
}
cout << "Total: " << total << endl;
```

The Loop-and-a-half Pattern

The loop-and-a-half is a kind of loop that is available in some language (like Ada's **`Exit When`**), but which must be simulated in C and C++, by using ***`if`*** and **`break`**. This is another way to write a sentinel loop.

- Write an endless loop (or a **`while`** loop with a necessary condition).
- Add in ***`if`*** statement inside the loop which checks the **sentinel**.
- If you find the sentinel, use **`break`**, which has the effect of immediately terminating the nearest enclosing loop.

The loop-and-a-half pattern has the advantage that it **follows the natural structure**: the **read-until-sentinel** pattern:

```
While True
   Prompt user and read value
   If value is the sentinel then
      break out of the loop
   Process the value
```

Note that this is an endless loop, where the only way to exit is by executing the **`break`** statement. Here's the same problem as on the previous page, using the **loop-an-a-half pattern**. You may want to look back and compare them.

```
while (true)                    // Endless loop
{
    cout << "> ";               // Prompt and read item
    cin >> value;
    if (value == 0) { break; }  // Sentinel? Leave loop
    total += value;             // No sentinel? Process
}
```

The Flag-controlled Pattern

A third way to implement the read-until-sentinel pattern is to use a flag-controlled loop, where you introduce an additional *Boolean* variable just before the loop starts and set it to **`false`**. Inside the loop you read a data value and check the sentinel, just as in the loop-and-a-half.

Instead of a **`break`** statement, set your flag variable to **`true`** when the sentinel is read. Otherwise, you process that data value as normal:

```
Set finished to false   // Boolean control flag
while not finished
  read the value
  if value is the sentinel then
    set finished to true
  else
    process the variable
```

As we've done with the other two methods, here is the same program implemented as a **flag-controlled** sentinel loop:

```
bool finished{false};       // control flag
while (! finished)
{
    cout << "> ";               // Prompt and read item
    cin >> value;
    if (value == 0)
    {
        finished = true;
    }
    else
    {
        total += value;
    }
}
```

Which of these thre versions **should** you use? Eric Roberts, a professor at Stanford for many years, suggests that empirical studies demonstrate that students are more **likely to write correct programs if they use the loop-and-a-half version** than if they are forced to use some other strategy. If you're interested, follow the link to read Roberts' paper.

Validating Data

When you read a value from `cin`, it is possible that the input may fail because the user entered invalid data. For instance:

```
cout << "Enter a number: ";
int n;
cin >> n;
cout << n << endl;
```

Suppose that the user types in **`one`** when asked to enter a number. Here's what happens:

1. The **`cin`** object enters a **failed** state and will stop accepting any more input.
2. The variable **`n`** will be set to **`0`**.

You can check for success by calling the member function **`fail()`** or by simply using a regular ***`if`*** statement. Here's a fragment that shows how to use ***`if`***:

```
int n;
if (cin >> n) { cout << n << endl; }
else { cout << "Invalid input" << endl; }
```

And, here's a fragment which explicitly calles the **`fail()`** member function:

```
int n;
cin >> n;
if (cin.fail()) { cout << "Invalid input" << endl; }
else  { cout << n << endl; }
```

Recovering

Inside a sentinel loop, you'd like to **recover** if the user inadvertently entered bad data.

1. Call **`cin.clear()`** to allow **`cin`** to start accepting data once again.
2. **Consume** the bad data by creating a **`string`** variable and reading it.

```
while (true)                    // Endless loop
{
    cout << "> ";                // Prompt and read item
    if (cin >> value) {
        if (value == 0) { break; }  // Sentinel? Leave loop
        total += value;             // No sentinel? Process
    }
    else {
        cin.clear();                // Clear the fail flag
        string bad_data;            // store the bad data
        in >> bad_data;             // read it and ignore it
    }
}
```