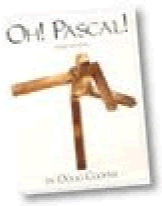
Chapter

7

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 have 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** that 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, 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.

```
// Before the loop
Test Condition
{
    // Loop actions
}
// After the loop
```

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

- actions that occur before the loop is encountered
- a **test** that determines the loop's bound
- 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.

Loop Mechanics

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 can quit 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
```

Bounds Preconditions

The bounds make sure that it is possible to get **out** of the loop. Next you have to that make it possible to **enter** the loop. Step 1 asks **"How do I get out?"**, while Step 2 asks: "**How do I get into the loop?** Look at your 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 code, your program 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 of them to some meaningful state.

```
// Step 2 : The bounds precondition
pos <- 0
letter <- str.at(pos)
While the 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.)

Advance 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.
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, **end-lessly**. Endless (infinite) loops are common errors.
3. To avoid, be sure the statements inside the loop body **change something** tested in the loop bounds. Here, just store the next character, like this:

```
pos <- 0
letter <- str.at(0)
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 "translated it" to a programming language, 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 goal 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 is produced?"** Then, **create and initialize variables** to store that information.

```
// Step 4: The Goal Preconditions
counter <- 0
pos <- 0
letter <- str.at(0)
While letter is not a period
{
    pos <- pos + 1
    letter <- str.at(pos)
}
// After the Loop
```

The Operations or Actions

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 precondition** in some way.

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.

```
counter <- 0
pos <- 0
letter <- str.at(0)
While letter is not a period
{
    // Step 5: The Goal Operation or Action
    counter <- counter + 1
    pos <- pos + 1
    letter <- str.at(pos)
}
// After the Loop
```

The Postcondition

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

In this 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:

```
counter <- 0
pos <- 0
letter <- str.at(0)
While letter is not a period
{
    counter <- counter + 1
    pos <- pos + 1
    letter <- str.at(pos)
}
// Step 6: The Loop Postcondition (reached the goal?)
counter <- counter + 1 // the period
// The variable counter contains the goal
```

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

Some Important Details

The problem said that the input would have a period if characters were entered. 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 != ""
{
   counter <- 0
   . . .
}
If counter is -1 the goal not reached
Else counter contains the goal
```

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

Necessary & Intentional Bounds

If we **change the problem** so 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 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.

Applying Loop Patterns

Now that you have a process for writing loops correctly the first time, let's put that process to work by completing `H06` together. In your CS50 IDE you'll find a problem similar to the short CodingBat style exercises you probably met in CS 170.

Here's the problem.

> *Write the function* `sumNums()`*, which, when given a* `string`*, returns the sum of the numbers appearing in the string, ignoring all other characters. A number is one, or more digit characters in a row.*

Here are some examples:

- `sumNums("abc123xyz")→123`
- `sumNums("aa11b33")→44`
- `sumNums("7 11")→18`

While you have enough syntactical information to solve the problem at this point, many of you won't really have any idea where to start. Let's go ahead and walk through the problem together.

Write the Stub or Skeleton

Always start by writing the "skeleton" or **stub** for your function. Make sure that your code starts out **syntactically correct**, and then stays **that way**. This is one of the most important techniques you can learn as a programmer.

For a function, the first steps towards writing a skeleton or "stub" are to simply:

1. **Write a header that has the correct types for input and output.** As you can see from the problem statement, the input type is `string` and the output type is `int`. However, as you learned from the text, when we write functions, we'll always pass `string` objects **by reference**. Since we don't need to change the string, we'll use a constant reference.
2. **Add some braces** to supply a **body** for the function.

3. Create a return variable to hold the result. Look at the function return type to decide what type to make this variable. Initialize it to the "empty" value.

Add a `return` statement at the very end of your function.

Loop Mechanics

Now if you `make test` and at least one of your tests should pass. To solve the rest, we need a loop. The *`for`* loop is often used to **process strings**. The *`for`* loop, and the **asymmetric bounds pattern**, is ideal, because the subscripts use by strings and arrays all begin at `0`, and the last element is always found at `size() - 1`.

Here's where the loop-building plan you learned comes into play.

1. What is the **loop bounds** or stopping condition? When we have processed the last character in the *`string`*.
2. What are the **bounds preconditions**? The number of characters to process and an index to process the string. Both must be initialized.
3. What **advances the loop**? Increment the loop index.

Using this plan, we can translate this to code. The *`for`* loop is **designed** for this.

The Loop Bounds

```
for ( ? ; i < len ; ? ) . . .
```

Our loop will continue while the loop index, *`i`*, is less than the size of the *`string`*, *`len`*. We'll **assume** that the *`string`*, *`str`*, already exists.

The Bounds Precondition

```
for (size_t i{0}, len{str.size()} ; i < len ; ? )
```

Notice that in the *`for`* loop **initialization statement** we can define two variables, as long as they are both the same type. Putting the "mechanical" variables here keeps them from contaminating the rest of your program; they are local to the *`for`* loop. Here, instead of the = syntax used in the last chapter, I've used **uniform initialization**, which is slightly shorter.

If you don't remember *`size_t`*, then look back at the discussion of the `size()` member function in the previous lessons.

Advance the loop

When using the ***for*** loop, advancing the loop is just a matter of updating the loop index in the **loop update section** like this:

```
for (size_t i{0}, len{str.size()}; i < len; ++i)
    // process the characters here
```

C programmers often use **`i++`**. *Develop the habit of using prefix (*`++i`*) which will help when you use iterators and the C++ standard library containers.*

Now we're finished with **the Loop Mechanics**; let's tackle the goal next.

The Loop Goal

Remember that a loop **produces information**. Before the loop starts, we want to create variables to hold that information. In our case, we need a variable **`sum`** set to zero. We'll also need a temporary variable, which I'll call **`number`**, which will hold the "current" number, every time we encounter one. It should also be initialized with 0.

Here's what our code should look like now.

```
int sum{0};
int num{0};
for (size_t i{0}, len{str.size()}; i < len; ++i)
{
    // process the characters here
}
```

The Loop Operation

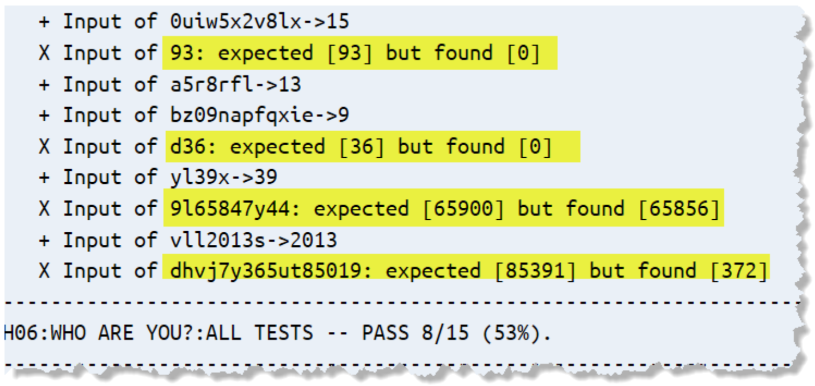
Let's spend a few moments **writing a plan** as a pseudocode.

```
sum <- 0
number <- 0
Grab the current character -> ch
If ch is a digit then
   digit <- ascii-to-decimal(ch)
   number <- number * 10
   number <- number + digit
Else
   sum <- sum + number
   number <- 0
```

Here are some notes on implementing this plan in C++.

- Use `str.at()` to grab the current character; store it in a variable
- Use `isdigit()` to see if the character is between `'0'` and `'9'` inclusive
- There is no ascii-to-decimal function. Instead, subtract the character `'0'` from `ch` and store the result in the variable `digit`. Note that the `ch` has an underlying `ASCII` code and the codes are sequential. If `ch` is `'0'` then subtracting `'0'` will store the binary number `0` in `digit`. If the character is `'1'`, then subtracting `'0'` will leave the binary number `1`.
- Multiply the current value of `number` by `10`, and then add the `digit`. For instance, if `number` has the value `2` and `digit` has the value `5`, then `number * 10` -> `20` and adding `5` leaves `number` with `25`, which is correct.
- When you encounter something that isn't a digit, then add the current value of `number` to `sum`. Then, set `number` back to `0` for the next iteration.

Go ahead and implement this code. Then do `make test`. You'll find that some of the tests pass, but others don't. Here's one possible run.



```
  + Input of 0uiw5x2v8lx->15
  X Input of 93: expected [93] but found [0]
  + Input of a5r8rfl->13
  + Input of bz09napfqxie->9
  X Input of d36: expected [36] but found [0]
  + Input of yl39x->39
  X Input of 9l65847y44: expected [65900] but found [65856]
  + Input of vll2013s->2013
  X Input of dhvj7y365ut85019: expected [85391] but found [372]
-------------------------------------------------------------------
H06:WHO ARE YOU?:ALL TESTS -- PASS 8/15 (53%).
-------------------------------------------------------------------
```

Do you see any way that the failing tests are different than those that pass? It takes a sharp eye, but if you look closely, all the tests that pass end in a letter, and all the tests that fail end in a digit. Which means if we are processing a number **when our loop ends**, we never add that number to the accumulator.

Our loop has **satisfied the bounds condition, but hasn't reached its goal.** That's the purpose of Step 6 in our loop-building strategy, the Goal Postcondition.

The Goal Postcondition

After the loop is over, one of two things can be true. If the last character processed is a letter or other non-digit, then the variable `number` contains the value `0`. If the last character was a digit, however, then `number` contains a non-zero value, which needs to be added to the `sum`. Of course, since adding zero has no effect on the `sum`, we can just add `number` to the `sum` one last time after the loop ends, and finally reach our goal.

Go ahead and do that. Then do `make test`. If you haven't made any mistakes, submit your code and you're done with this assignment.

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
    a. Make sure you submit the assignment using **`make submit`**.
    b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.