Homework

5

Strings, Loops, Numbers

A common use of the `for` loop is to process strings and arrays. 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`.

In this assignment, you'll find a loop problem from the hand of Nick Parlante[1] similar to the short CodingBat[2] style programs you met in CS 170. These are similar in style and form to the problems you'll be asked to solve for **Programming Exam** 3.

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.

---

[1] http://cs.stanford.edu/people/nick/

[2] http://codingbat.com/java

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`, at least one of your tests should pass. To solve the rest, you 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 C++ code using the *`for`* loop.

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. Feel free to use the **=** syntax if it is more comfortable.

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++**. If you develop the habit of using prefix (**++i**), it will help when you start to 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
for each character in str
   Set 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.