Homework

14

A Console Input Library

Since we've been talking about streams, functions, and libraries, let's get some practice implementing the functions in a library for simple input and output. Based on the [`simpio.h`] library that is part of the **Stanford cslib C++ libraries** and the **Harvard cs50-c libraries**, it is a little different from both of them.

An Interactive Console Input Library

The library focuses entirely on **interactive** (**console-based**) input. Instead of this:

```cpp
int n;
cout << "Enter an integer: ";
cin >> n;
```

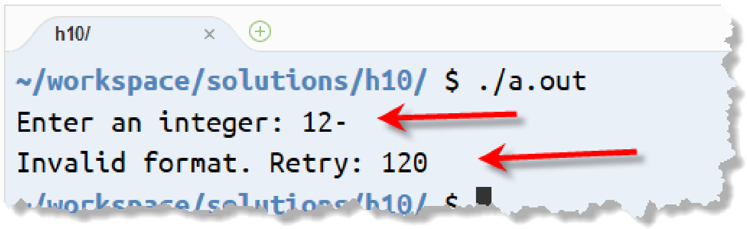
Stream-based input.

With the library, you can write this instead:

```cpp
int n = getInt("Enter an integer: ");
```

Input-library example.

While considerably shorter, it, more importantly, **checks the user's input for errors**.



```
~/workspace/solutions/h10/ $ ./a.out
Enter an integer: 12-
Invalid format. Retry: 120
~/workspace/solutions/h10/ $
```

If the user intends to type `120` but mistakenly types a minus sign in place of the zero, `getInt()` will notice that the input is not a legal integer and give the user another chance to enter the correct value.

The `>>` operator, by contrast, does not check for this type of error. If the user enters `12-` in response to the classic input statement, then the variable `n` would get the value `12`, leaving the minus sign in the input stream waiting for the next input operation.

Other Functions

In addition to `getInt()`, our library will also contain `getReal()` for reading a floating-point (`double`) value, `getLine()` for reading an entire line as a string while supplying an optional prompt, and `getYN()` for asking a yes/no question with a prompt.

→ `string getLine(const string& prompt)`: first prompts the user and then-reads a line of text from `cin`, returning that line as a `string`. It is similar to the built-in `getline()` function except that it **displays a prompt** (if provided) and returns a string rather than modifying its argument. If there is a prompt and it does not end in a space, a space is added.

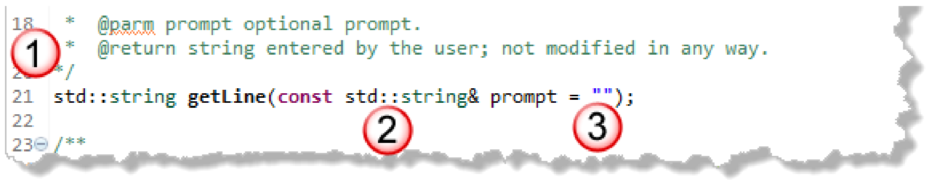
→ `int getInt(const string& prompt)`: reads **a complete line** and then converts it to an integer. If the conversion succeeds, the integer value is returned. If the argument is not a legal integer or if extraneous characters (**other than whitespace**) appear in the string, the user is given a chance to reenter the value. The `prompt` argument is optional and is passed to `getLine()`.

→ `double getReal(const string& prompt)`: works like `getInt()` except it returns a `double`.

→ `bool getYN(const string& prompt)`: works similarly, except it looks for any response starting with `'y'` or `'n'`, case insensitive.

1. Declaring the Interface

Add the prototypes and complete the comments in the interface. Here's my code for `getLine()`. The explanatory comments were copied from the instructions (above).



```
 *  @parm prompt optional prompt.
 *  @return string entered by the user; not modified in any way.
 */
std::string getLine(const std::string& prompt = "");

/**
```

Three important things to remember:

1. For **any library type** used in a header you include the necessary header file and then must fully qualify **every type name** in the header

2. Think about which way **the data is flowing**. An input library-type parameter, is always defined as a **constant reference**. Output parameters are defined as regular references.
3. The `prompt` is an **optional** or **default argument**. For default arguments, supply the default value **in the prototype**. When we you out the function, remove the default value.

Run `make test` at this point. You should ONLY get linker errors. If you get any syntax errors, then please fix them.

2. Stub the Implementation

To stub out **the implementation**, copy the prototypes from the header file to the implementation file. Remembering to remove the semicolons at the end of each prototype and remove the default values from each of the optional parameters.

You may also remove the fully-qualified type names from the parameters and return type, but you don't need to do that. (I just think it looks neater, so I do it.)

You should be able to build and test your project. Although most of the tests will fail, you should not have any compiler or linker errors.

3. Implementation Notes

The `getLine()` function is easiest. First, print the prompt, if supplied. Then, use the `string::getline()` function to actually read the input. The tricky part is checking if the `prompt` ends in a space.

- **Was a prompt supplied?** Use the `string::empty()` member function to check. If it wasn't supplied, remember that you still need to read the input.
- If a prompt was supplied, **is the last character a space**? Use the `back()` member function (or the more verbose `str.at(str.size() - 1)`).
- To see **if the character is a space** (of any sort) using the `isspace()` function from `<cctype>`, which you must include along with the others. If it is not a space, then add a space after the prompt.

Now, users can call **your** `getLine()` function in all three ways.

```
string s1 = getLine("What's your name?"); // add space
string s2 = getLine("Age: "); // no space needed
string s3 = getLine(); // no prompt
```

Try completing the function and then run the tests to see if you can pass the first section. If you have difficulty, ask questions on Piazza or come to my office hours.

The Number Input Functions

Here's the plan for both `getInt()` and `getReal()`:

```
Read an entire line from the user (use your getLine)
Scan the line for an int (or double)
   - skip any spaces before or after
If the scan failed, or, if there are any characters left
   - Display an error message and read another line
   - Go back to line 2
Otherwise, return the scanned value
```

Copy this as an implementation comment for `getInt()` and then think about it.

The Basic Function Skeleton

The first thing you need is **a loop**. Exit the loop if you have succeeded and print a message and repeat if you fail. An **endless loop** is probably the easiest way to write this.

Before the loop, read a line of input using `getLine()`, passing the prompt. **Only do this once**; if there is an error, you'll print an error message instead. Once you have a line of text, convert it to an `int` by using the `istringstream` class.

Here's the modified pseudocode for this:

```
Prompt and read input from the user
While True
    If input is not empty Then
        Convert input into istringstream in
        Create a local numeric variable n
        Read from in into n
        If there are no errors, return n
   Display an error message, clear input read another
```

Validating the Input

To tell if you **actually** read a number, check two things:

- If the string cannot be parsed as an number, `in.fail()` will return `true`
- If the `string` begins with a number but contains additional characters, `stream.eof()` will be `false`

Using this information, we can return **n** when `in.eof() && ! in.fail()`.

Extra Whitespace

According to the specification, you allow whitespace either before or after the value. The first `>>` operator automatically skips over any whitespace characters that appear before the value. To **read whitespace that follows** you need to add the line:

```
in >> ws;
```

The `ws` manipulator consumes any whitespace up to the end of line, thereby ensuring that the stream will be positioned correctly at the end of the line if the input is correctly formatted. Here's the modified pseudocode with validation added:

```
Prompt and read input from the user
While True
    If input is not empty Then
        Convert input into istringstream in
        Create a local numeric variable n
        Read from in into n
        If in.eof and not in.fail return n
        Strip trailing whitespace from n
        If in.eof and not in.fail return n
   Display an error message, clear input read another
```

Why you can't just check for the number and the whitespace in one read, instead of two? You can! And. it will work on most platforms including Linux and Windows. However, the standard library used on the Mac will fail if the input does not have any trailing whitespace. The design I've used here works correctly everywhere.

The Yes/No Function

After the number functions, `getYN()` should be fairly simple. Follow the same pattern as with the numeric input functions. Your validation is:

1. Make sure there is a first character; if not, try again
2. Compare against `'Y'`, `'y'`, `'N'` and `'n'`. If it is not one of those, try again.
3. If it is `'Y'` or `'y'`, return `true`, otherwise `false`.

Now all tests should pass. Once you've gotten 100% on all the tests go ahead and submit your code. If you have difficulty, ask on Piazza or come to my office hours.