CSE333 Exercise 0

Out: Wednesday, September 24
Due: Friday, September 26 by **10 am**
Rating: 3 (note)

Goals

- Write a C Program from scratch.
- Write code that utilizes command-line arguments and C-string parsing.
- Write code that uses function declarations.
- Read C documentation to learn how C standard library functions behave.
- Use the linter and valgrind.

Background

One way to estimate π is to use the following infinite series, which was discovered by Nilakantha in the 15th century:

```
π = 3 + (4 / (2 x 3 x 4)) - (4 / (4 x 5 x 6)) + (4 / (6 x 7 x 8)) - ...
```

Breaking the series down:

- The zero'th term in the series is: 3
- The first term in the series is: `+(4 / (2 x 3 x 4))`
- The second term in the series is: `-(4 / (4 x 5 x 6))`
- The n'th term in the series is: $(-1)^{(n+1)} \cdot (4 / (2n \cdot (2n+1) \cdot (2n+2)))$

Problem Description

Write a C program that estimates π by adding together terms 0 through n, ***inclusive***, in the Nilakantha series, and prints out that estimate to *20 decimal places*.

- "n" is provided to your program as a command-line argument, which you should attempt to parse as an `int`.
- You should compile your program into an executable called `ex0`.
- Example executions and their resulting outputs are shown below. *Make sure that your submission matches these outputs exactly!*

```
$ gcc -Wall -g -std=c17 -o ex0 ex0.c
$ ls
ex0     ex0.c
$ ./ex0 100
Our estimate of Pi is 3.14159241097198238535
$
```

Implementation Notes

Libraries

As part of this exercise you will need to explore some of the basic C libraries in order to convert the command-line argument (the string of digit characters) to an `int` value and print the result. The nav bar above links you to www.cplusplus.com/reference (http://www.cplusplus.com/reference) as the "C/C++ Reference", which contains information about the C and C++ libraries (ignore the C++ info for now).

- In particular, you will find it useful to look at (1) the `stdio.h` library for basic input and output, (2) `string.h` for handling C-strings (*i.e.*, null-terminated arrays of characters), and (3) `stdlib.h` for some useful functions.
- There are multiple ways to parse (convert) strings of digit characters to integer values, including `sscanf`, `atoi`, and others; think about the tradeoffs and differences between them.
- You are allowed to use any standard library function that does not require additional compilation options (see Submission details below). In particular, you may not use functions from the `math.h` library.

User Input

You should handle various inputs from the user, which may be in an unexpected format. For each input, you should take some time to reason through your options for handling it gracefully (*i.e.*, without unexpectedly crashing), decide which one seems "best", and document your decision in your code. Note that you do not need to give a fully detailed explanation; short well-written comments are fine. Example unexpected inputs for this exercise include, but are not limited to:

-
```
$ ./ex0
```

- ```
  $ ./ex0 hello
  ```
- ```
  $ ./ex0 0.05432
  ```
- ```
  $ ./ex0 10000000 3234
  ```

For simplicity, one case you can ignore is when someone inputs an integer that is too large to fit into an `int` data type (*e.g.*, 2^{351} is too big to fit into 32 bits).

Arithmetic

To avoid floating-point arithmetic errors, please make sure your code adheres to the following rules:

1. Use C's `double` type for intermediate and final results.
2. Sum the series in order from term 0 to term n (and not the other way around).

Style Focus

General

As a starting point, get comfortable with the C best practices mentioned in the first lecture, particularly focusing on **program layout** and **function ordering**.

Other style issues to keep in mind, even though not discussed in lecture, include: indenting your code consistently, factoring out code into helper functions when appropriate, and writing an adequate amount of useful comments.

User Input

You will also be expected to be **robust in handling user input**, which includes handling unexpected inputs. Refer to the section above for more details.

Tools

Make sure that you run the linter (`./cpplint.py --clint ex0.c`) and valgrind (`valgrind ./ex0 <input>`) on your code and resolve any indicated issues before submitting.

- See the cpplint page (../cpplint.html#download) for how to obtain a copy of the linter.
- **For this exercise only:** you may ignore the error "sscanf can be ok, but is slow and can overflow buffers".
- When you have fixed all linter errors, you should see the following output from `clint.py` :

  ```
  $ ./cpplint.py --clint ex0.c
  Done processing ex0.c
  ```
  ```

- For *each* choice of `<input>`, you should be looking to see the following line of output from `valgrind`:

```
$ valgrind ./ex0 <input>
...
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Submission

You will submit: `ex0.c`.

For full credit, your code must:

- Compile without errors or warnings on CSE Linux machines (lab workstations, `attu`, or CSE home VM).
- Have no runtime errors, memory leaks, or memory errors (`gcc` and `valgrind`).
- Be contained in the file listed above that compiles with the command:

```
$ gcc -Wall -g -std=c17 -o ex0 ex0.c
```

  - `math.h` is disallowed because it won't work without the addition of the `-lm` option.
- Have a comment at the top of your `.c` file with your name(s) and CSE or UW email address(es).
- Be pretty: the formatting, modularization, variable and function names, commenting, and so on should be consistent with class style guidelines. Additionally, the linter shouldn't have any complaints about your code (`cpplint.py --clint`).
- Be robust: your code should deal with hard-to-handle/edge cases and bogus user input (if there are any) gracefully.

Submit your code on Gradescope (../submit.php).