Assignment 6: Deriving Pi

Topics and References

- Functions. This page contains a detailed description of functions. Chapter 9 of the text has information related to declaring and defining functions and pass-by-value semantics.
- Iteration structures. See this page for details of the `while` statement, this page for details of the `for` statement, and this page for details of the `do ... while` statement. Chapter 6 of the text contains information for iteration statements.

Task: Computing Pi

The purpose of today's exercise is to understand the meanings and terminology associated with functions: declaration, definition, arguments, parameters, and pass-by-value semantics. You'll do this by declaring and defining three functions that will approximate the value of pi where pi is defined as the ratio of a circle's circumference to its diameter.

Implementation Tips

You're expected to implement two functions in `pi.c` and without completely defining these three functions, you cannot successfully compile, link, and execute the program. This means that you cannot concentrate on implementing, testing, and verifying only one function at a time because you need all three to be implemented to just compile and link. It is also possible that you implement the first function incorrectly and transfer these errors to the other two functions. Later, you're faced with the task of trying to debug three incorrect functions. Is there a better alternative?

The better alternative consists of implementing *stub functions* for each of `main`'s sub-functions. A stub is a skeleton of a function that is called and immediately returns. It is syntactically correct - it takes the correct parameters and returns the proper values (although you may need to drop the `-Werror` flag to successfully compile in certain cases). Although a stub is a complete function, it does nothing other than to establish and verify the linkage between the caller and itself. But this is a very important part of coding, testing, and verifying a program. At this point, the program should be compiled, linked, and run. Chances are that you'll find some problems related to syntax such as missing semicolons or errors between function prototype declarations and the function definitions. Before you continue with the program, you should correct these problems. After implementing, testing, and verifying the first function, you should replace the second function's stub with the actual code that implements the required functionality. This second implementation is then completely tested and verified before moving to the third function. Remember to turn on the `-Werror` flag if you have previously turned it off for compiling stub functions.

Here are some more tips and common programming errors related to functions.

1. Several possible errors are related to passing parameters.
    1. It is a compile error if the types in the prototype declaration and function definition are incompatible. For example, the types in the following code fragment are incompatible:

```
// function prototype declaration
double divide(int dividend, int divisor);

// function definition
double divide(float dividend, float divisor) {
  // perform some sort of division here ...
}
```

2. It is a compile-time error to have a different number of arguments in the function call than there are parameters in the prototype.
3. It is a logic error if you code the parameters in the wrong order. Their meaning will be inconsistent in the called function. For example, in the following code fragment, the types are the same but the meaning of the variables is reversed:

```
// function prototype declaration
double divide(float dividend, float divisor);

// function definition
double divide(float divisor, float dividend) {
  // perform some sort of division here ...
}
```

2. It is a compile-time error to define local variables with the same identifiers as formal parameters, as shown below:

```
// function prototype declaration
double divide(float dividend, float divisor);

// function definition
double divide(float divisor, float dividend) {
  // local definitions
  float dividend;

  // perform some sort of division here ...
}
```

3. Using a `void` return with a function that expects a return value or using a return value with a function that expects a `void` return is a compile-time error.
4. Each parameter's type must be individually specified; you cannot use multiple definitions like you can in variables. For example, the following is a compile-time error because `y` doesn't have a type:

```
double bad(float x, y);
```

5. Forgetting the semicolon at the end of a function prototype is a compile-time error.

```
double badder(float x, float y)
```

6. Similarly, using a semicolon at the end of the header in a function definition is a compile-time error.

```
double baddest(float x, float y);
{
  // do bad stuff here ...
}
```

7. It is most likely a logic error to call a function from within itself or one of its called functions. This is known as recursion, and its correct use is covered later in the semester.
8. It is a compile-time error to attempt to define a function within the body of another function.

```
double hilarious(float x, float y) {
  float z = x + y;
  double naughty(float z) {
    // do naughty stuff here ...
  }
  // do hilarious stuff here ...
}
```

9. It is a run-time error to code a function call without the parentheses, even when the function has no parameters.

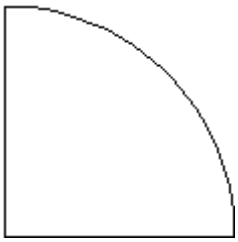
```
// function definition
void print_hello() {
  printf("Hello World\n");
}

// somewhere inside another function, say main() ...
int main() {
  /*
  the name of a function evaluates to the address of the first
  instruction of the function
  */
  print_hello;
  // use the function call operator () to invoke a function
  print_hello();
}
```

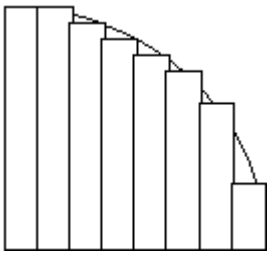
10. It is a compile-time error if the type of data in the `return` statement doesn't match the function return type.

Calculus Algorithm

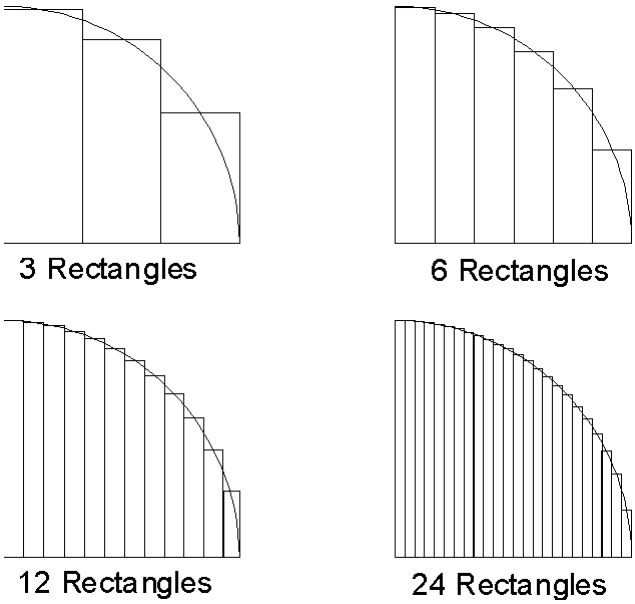
Consider a quarter circle with radius `r=2` units (the circle must have radius 2 and no other value for this algorithm) shown in the picture below.



From circle area formula, `area = pi * r*r`, so the area of a quarter circle with radius 2 units is `pi * 2*2 / 4` which is pi square units. All that is required to computationally determine pi's value is to compute the quarter circle's area with radius 2 units. The calculus algorithm computes the quarter circle area by dividing it into a series of rectangles. These rectangles have similar width but different heights with a particular rectangle's height chosen such that the circle's circumference passes through the midpoint of the rectangle's top. This is shown in the picture below:



Increasing the number of rectangles provides a closer approximation of the circle's area (and a better value for pi). One consequence of increasing the number of rectangles is that each of the rectangles is thinner:



As the number of rectangles increases to infinity, each rectangle's width decreases to zero, and the summation of the areas of all these rectangles will provide you with pi's value.

How are you going to compute the area of each individual rectangle? Recall that given a rectangle's width `w` and height `h`, its area is `area = w * h`. Assume there are `n` rectangles. As seen in the pictures, each of these `n` rectangles will have the same width `w` but different heights - rectangles closer to the circle's center are taller than those near the edge. Based on the assumption that the circle's circumference passes through the midpoint of a rectangle's top edge, the height `h` of a rectangle whose midpoint is located `x` units from the circle center can be computed using the Pythagorean theorem as `h = sqrt(2 * 2 - x * x)` since the circle's radius is 2 units.

Finally, you've an algorithm to approximately compute a quarter circle's area with radius 2 units. First, divide this quarter circle into a large number of rectangles having equivalent widths but different heights. Second, use a `for` or `while` statement to iterate through these rectangles and sum their areas. The larger the count of rectangles, the closer is the sum to the mathematical value of pi.

Implement the above version of the calculus algorithm to calculate pi's value in a function `calculus_pi` with prototype

```c
double calculus_pi(int slices);
```

where function parameter `slices` represents the count of rectangles. The function returns the area of the summed rectangles. Use standard C library function `sqrt` declared in `<math.h>` to compute `h = sqrt(r * r - x * x)`.

Leibniz Algorithm

The second function derived by Leibniz uses an infinite series of additions and subtraction to approximate as follows: $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$

You will require a `for` or a `while` statement to sum the elements of a finite number of elements in the series. As the number of terms in the series increases, the approximation will be closer to the value of pi. After the summation of the terms, the value must be multiplied by 4 to arrive at the final approximation of pi.

Implement the Leibniz algorithm to calculate the value of pi in a function with prototype

```c
double leibniz_pi(int terms);
```

where function parameter `terms` represents the number of elements in the series. The function returns the approximate value of pi.

Comparisons

The algorithms can be compared by having function `main` print the pi values. The output would look like this:

```
calculus_pi(1) returned 3.464
calculus_pi(10) returned 3.152
calculus_pi(100) returned 3.142
leibniz_pi(1) returned 4.000
leibniz_pi(10) returned 3.042
leibniz_pi(100) returned 3.132

```

Remember the `'\n'` at the end of each output line.

Deliverables and Submission

1. You are given file `driverpi.c` which includes function `main`. You'll not be submitting `driverpi.c`. This means that any changes you make to `driverpi.c` will not be seen by the grader. Repeat: The grader will use its own equivalent of `driverpi.c` and not yours. In fact, you must not include `driverpi.c` for submission.
2. Notice that `driverpi.c` includes header file `pi.h`. This header file will contain the *declarations* of the functions that you're to implement. You must author this file. Without this file, you (and the grader) will not be able to successfully compile and link your submission. Start by adding a file header that looks like this:

```
/*!
@file     highbury-lane
@author   Nicolas Pepe  (nicolas.pepe@digipen.edu)
          Rob Holding   (rob.holding@digipen.edu)
          Mezut Ozil    (mezut.ozil@digipen.edu)
@course   CS 999
@section  ???
@tutorial Tutorial 1
@date     02/02/2020
@brief    This file contains code that puts a salmon on a cedar
          plank and smokes the fish for three hours.
*//*_____________________________________________________________________*/
```

Since this file is the interface that other programmers will use to understand your pi library, you must provide *function-level* documentation for every function. This documentation should consist of a description of the function, the inputs, and return value. This will provide your clients an opportunity to understand the details necessary for them to use functions from your library. In general, recall that your clients will not have access to the source file where you defined these functions (because they're your intellectual property and you don't want others to steal that property). Instead, you will provide clients the header file and a binary library file containing your implementation in machine language. An example of function-level documentation is provided below for a function `reverse32()` that returns the byte order of an unsigned 32-bit integer.

```
/*!
@brief Reverses the byte order of an unsigned 32-bit integral value.

This function takes as input a 32-bit integer and reverses the byte
order of the full word. That is, given the value 0x12345678, the
function will return the value 0x87654321.

@param word - the 32-bit word to be reversed.
@return the reversed 32-bit value of word
*//*_____________________________________________________________________*/
unsigned int reverse32(unsigned int word);
```

3. You must author a file called `pi.c` that will contain the definitions of the functions that you're to implement. Obviously, these functions must be implemented exactly as you prototyped them in `pi.h`. As is required and necessary, this file must contain appropriate file and function headers.
4. The implementation must cleanly compile for C11 with the full suite of warning options. The command line to build this assignment will look like this:

```
gcc -std=c11 -pedantic-errors -Wall -Wextra -Wstrict-prototypes -Werror
driverpi.c pi.c -o pi.exe -lm
```

5. Your output must exactly match the example output above.
6. You must submit source file `pi.c` and header file `pi.h` in Moodle.

Short note on floating-point errors

In class lectures, we discussed the ability of integral types (`char`, `short`, `int`, `long`) to precisely represent values. For example, the `signed char` type can be used to precisely represent 256 "small" integral values in the range [-128, 127]. In contrast, floating-point types (`float`, `double`, `long double`) can only represent approximations of real values. Since there are infinite real values between any intervals of numbers, floating-point types cannot represent these infinite real values exactly. If you were asked to write the exact decimal notation for 1/3, you couldn't. There are an infinite number of 3s after the decimal point in that real number. Well, computers don't have an infinite amount of bits either. So, what to do? Computers use the IEEE 754 format to approximate real values as floating-point values. These approximations are close, but they are not exact, and therefore introduce errors into calculations involving floating-point types. Usually, this isn't a big deal. But, if you introduce many, many errors into the calculations, you will start to notice them.

Here's the idea and a hint at how to fix it. If you have a floating-point `x` initialized as follows:

```
double x = 1.0 / 3.0;
```

both of these statements are *supposed* to do the same thing:

```
x + x + x + x + x + x;
6 * x;
```

First realize that floating-point variable `x` is not initialized with the exact value of 1/3 but instead with an approximation to the exact value of 1/3. The error between the exact and approximate values is called a *rounding error*. Now, each time you introduce an expression `x + x`, you introduce a little more rounding error. Why? Because you are taking an approximation and adding to it an approximation to get a new floating-point number that is further approximated. If you again add another approximate value to the previous approximate value, you will get an approximation with a larger rounding error. If you add enough `x`'s together, you will see that the result of all these additions will begin to drift away from the correct result much more than if you multiply the number once, as in `6 * x`.

The following code fragment has the same problem, although you may be fooled into thinking that you aren't adding `x` over and over again:

```
total = 0;
for (i = 0; i < 6; i++) {
  total += x;
}
```

Here's a code fragment that demonstrates the differences:

```
int count;
for (count = 10; count <= 100000000; count *= 10) {
  double x = 1.0 / count;
  double multiplied = x * count;
  double added = 0;
  int i;

  for (i = 0; i < count; i++) {
    added += x;
  }

```

```
  printf("iterations: %i, x = %.8f\n", count, x);
  printf("    multiplied = %.14f\n", multiplied);
  printf("         added = %.14f\n", added);
  printf("\n");
}
```

The output clearly indicates that the rounding error is lower when performing multiplication such as `6 * x` rather than addition such as `x + x + x + x + x + x`:

```
iterations: 10, x = 0.10000000
    multiplied = 1.00000000000000
         added = 1.00000000000000

iterations: 100, x = 0.01000000
    multiplied = 1.00000000000000
         added = 1.00000000000000

iterations: 1000, x = 0.00100000
    multiplied = 1.00000000000000
         added = 1.00000000000000

iterations: 10000, x = 0.00010000
    multiplied = 1.00000000000000
         added = 0.99999999999991

iterations: 100000, x = 0.00001000
    multiplied = 1.00000000000000
         added = 0.99999999999808

iterations: 1000000, x = 0.00000100
    multiplied = 1.00000000000000
         added = 1.00000000000792

iterations: 10000000, x = 0.00000010
    multiplied = 1.00000000000000
         added = 0.99999999975017

iterations: 100000000, x = 0.00000001
    multiplied = 1.00000000000000
         added = 1.00000000228987

```

In closing, the above discussion should reiterate that you should always opt to implement code such as `6 * x` rather than `x + x + x + x + x + x+ x`.