Programming for computerteknologi
Hand-in Assignment Exercises

Week 7: Programming larger software projects

Exercise 1)

We have been given a task to program the Taylor series equation that we have talked about in class to implement an approximate sine function. Our function is able to calculate the sine function with precision of up to *n* Taylor series terms. The Taylor series equation looks like the following:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

To implement the equation, we need to implement both a factorial function and a power function. I have made such functions which looks like the following.

```
3   double factorial(int a) {
4       double fact = 1;
5       for(int i = 0; i < a; i++) {
6           fact = fact * (i + 1);
7       }
8       return fact;
9   }
```

```
11   double power(double a, int b) {
12       double pow = a;
13       for(int i = 1; i < b; i++) {
14           pow = pow * a;
15       }
16       return pow;
17   }
```

With this in mind, we can now make the following function that calculates the sine function using Taylor series terms.

```
19  double taylor_sine(double x, int n)
20  {
21      double finalSine = x;
22      int pow = 1;
23      int currentn = 2;
24
25      assert(0 < n);
26
27      for(int i = 0; i < n; i++) {
28          pow += 2;
29          if(n == 1) {
30              return x;
31          } else {
32              if(currentn % 2 == 0) {
33                  finalSine = finalSine - power(x,pow)/factorial(pow);
34              } else {
35                  finalSine = finalSine + power(x,pow)/factorial(pow);
36              }
37          }
38          currentn++;
39      }
40      return finalSine;
41  }
```

I have run a couple of tests with both different x-values as input and with different precision. The code looks like the following.

```c
int main(int argc, char **argv)
{
    double x;
    int n;

    x = 3.1415;
    n = 12;
    printf("\nsin(%f) = %f \n", x, sin(x));
    printf("my function with %d precision: %f \n\n", n, taylor_sine(x, n));

    x = 1.57075;
    n = 10;
    printf("\nsin(%f) = %f \n", x, sin(x));
    printf("my function with %d precision: %f \n\n", n, taylor_sine(x, n));

    x = 0;
    n = 1;
    printf("\nsin(%f) = %f \n", x, sin(x));
    printf("my function with %d precision: %f \n\n", n, taylor_sine(x, n));

    x = 21;
    n = 10;
    printf("\nsin(%f) = %f \n", x, sin(x));
    printf("my function with %d precision: %f \n\n", n, taylor_sine(x, n));

    x = 32;
    n = 95;
    printf("\nsin(%f) = %f \n", x, sin(x));
    printf("my function with %d precision: %f \n\n", n, taylor_sine(x, n));

    return 0;
}
```

```
sin(3.141500) = 0.000093
my function with 12 precision: 0.000093


sin(1.570750) = 1.000000
my function with 10 precision: 1.000000


sin(0.000000) = 0.000000
my function with 1 precision: 0.000000


sin(21.000000) = 0.836656
my function with 10 precision: 55708510.350769


sin(32.000000) = 0.551427
my function with 95 precision: 0.552733
```

As seen on the above output the bigger the input x-value is the result will vary more and more from the ANSI C sine function. The greater the input x-value is the more precise we need our Taylor series equation to be. When inserting 21 with a precision of 10 we get an incredible high number which means we need it to be a lot more precise. If we set the precision to 40 instead, we will get the same result as the ANSI C sine function.
If our input x-value on the other hand is 32 our function can't get the exact same result the ANSI C sine function even if we increase the precision. So, our function won't be perfect but it works on smaller input values.

```
sin(21.000000) = 0.836656
my function with 40 precision: 0.836656


sin(32.000000) = 0.551427
my function with 99 precision: 0.552733
```

Exercise 2)

We have been given a task to program a function that would implement a stack. A stack maintains the *last-in, first-out* (LIFO) order and consists of the following functions:

- Push(x,s) - Inserts an item at the top of the stack.
- Pop(s) - Removes an item from the top of the stack.
- Initialize(s) - Creates a stack.
- Full(s), Empty(s) - Test whether a stack is full or empty and therefore tests if the user can either push or pop more items to the stack.

When implementing a stack based on singly-linked lists as discussed in class it looks like the following functions:

```
3   void initialize(stack* s){
4     //implement initialize here
5     s->head = NULL;
6   }
```

```c
void push(int x, stack* s){
  //implement push here
  node *new;
  new = (node*) malloc(sizeof(node));
  new->data = x;
  if (s->head == NULL) {
    new->next = NULL;
  } else {
    new->next = s->head;
  }
  s->head = new;
}
```

```c
int pop(stack* s){
  // implement pop here
  if(s->head == NULL) {
    printf("\nEMPTY STACK\n");
  }
  else {
    node *t = s->head;
    int y = s->head->data;
    s->head = s->head->next;
    free(t);
    return y;
  }
  return 0;
}
```

```c
void empty(stack* s)
{
  if(s->head == NULL) {
    printf("Stack: EMPTY");
  } else {
    printf("Stack: NOT EMPTY");
  }
}
```

The function Full isn't necessary since the list doesn't have a maximum stack size which means that it can't be full.

When testing the stack function, we can take a look at the given test cases since they cover all the assertions, we need to pass to call our function successful. REQUIRE(sb1.head == sb0.head);

```
===============================================================================
All tests passed (10 assertions in 2 test cases)
```

The first testcase ensures that the stack s is empty after initializing it.

```
stack s;
initialize(&s);
REQUIRE(s.head == NULL);
```

The next testcase ensures that the stack s is exactly the same after calling the functions push(x, s) and pop(s). This can be seen at REQUIRE(sb1.head == sb0.head);

```
stack sb0;
initialize(&sb0);
stack sb1 = sb0;
int x = 5;
push(x, &sb0);
REQUIRE(sb1.head != sb0.head);
int y = pop(&sb0);
REQUIRE(x == y);
REQUIRE(sb1.head == sb0.head);
```

The last testcase ensures that after executing push functions and pop functions, the variables pushed is the same as the variables popped in the right order.

```
stack sc0;
initialize(&sc0);
stack sc1 = sc0;
int x0 = 1;
int x1 = 2;
push(x0, &sc0);
push(x1, &sc0);
int y0 = pop(&sc0);
int y1 = pop(&sc0);
REQUIRE(x1 == y0);
REQUIRE(x0 == y1);
REQUIRE(sc0.head == sc1.head);
```