Programmering 7

Opg. 1

Code answer In the lecture we discussed how we could implement an approximate sine function using the Taylor series equation:

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

a) Write this function in a C program so that it calculates the sine function with precision up to $n$ Taylor series terms, e.g. the above example shows 4 terms. Your sine function should accept both the $x$ value and $n$ precision as input. The signature of your function should be:

```c
double taylor_sine(double x, int n) {
  // implement your function here
  return -1.0;
}
```

Write your Taylor series implementation of sine as a library consisting of: a header file and a source file (with no main() function).

```
> .devcontainer
> .vscode
> tests
.gitignore
assignment-7.pdf
CMakeLists.txt
Makefile
README.md
stack.c
stack.h
taylor_sine.c        M
taylor_sine.exe
taylor_sine.h        1, M
```

```c
#pragma once

double taylor_sine(double x, int n)
{
    double sum = x;
    double fact = 1;
    int sign = -1;
    // implement your function
for (int i = 3; i <= n; i += 2)
{
    fact = fact * i * (i-1);
    sum = sum + sign *pow(x,i)/fact;
    sign = sign * -1;
}
    return sum;
}
```

b) Write some tests for different values of $x$ (try both small and large input values), and compare your function output with the ANSI C sin function.

```
value of n:5
Value of x:2
sin(2.00) = 0.933333
```

```
value of n:3
Value of x:90
sin(90.00) = -121410.000000
```

```
value of n:1
Value of x:0
sin(0.00) = 0.000000
```

Write your test program (which will have a **main()** function) separately from your library, i.e. **create a new file** where you only include the library header file. Compile your test program by *linking* with your Taylor Sine library.

(c) Answer the following questions using your test program, and please **provide your answers as comments in your test program**: Which intervals of input $x$ did your function give a similar result to the ANSI C sin function? What impact did increasing the precision have (i.e. increasing the number of Taylor series terms)?

```
int main()
{
  int n;
  double x, sine;

  printf("value of n:");
  scanf("%d", &n);
  printf("Value of x:");
  scanf("%lf", &x);

//kalder funktionen
sine = taylor_sine(x, n);
printf("sin(%.2f) = %.6f\n", x, sine);//vores funktion slutter ved n

//ANSI C sin funktion
printf("sin(x) = %f\n", sin(x)); //Deres funktion fortsætter uendelig
//meget mere præcis ved højere tal som f.eks. x = 90

return 0;
}
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
value of n:3
Value of x:90
sin(90.00) = -121410.000000
sin(x) = 0.893997
```

2)

Code answer Stacks are containers where items are retrieved according to the order of insertion, independent of content. Stacks maintain *last-in, first-out order (LIFO)*. The abstract operations on a stack include:

| **Function** | **Description** |
|---|---|
| `push(x, s)` | Insert item `x` at the top of stack `s`. |
| `pop(s)` | Return (and remove) the top item of stack $s$ |
| `initialize(s)` | Create an empty stack |
| `full(s)`, `empty(s)` | Test whether the stack can accept more pushes or pops, respectively. *There is a trick to this, see if you can spot it!* |

Note that there is no element search operation defined on standard stacks. Defining these abstract operations enables us to build a stack module to use and reuse without knowing the details of the implementation. The easiest implementation uses an array with an index variable to represent the top of the stack. An alternative implementation, using linked lists, is better because it can't overflow.

Stacks naturally model piles of objects, such as dinner plates. After a new plate is washed, it is pushed on the top of the stack. When someone is hungry, a clean plate is popped off the top. A stack is an appropriate data structure for this task since the plates don't care which one is used next. Thus one important application of stacks is whenever **order doesn't matter**, because stacks are particularly simple containers to implement.

(a) Implement a stack based on **singly-linked lists** as discussed in the lecture.

```c
#include "stack.h"
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <assert.h>

void initialize(stack *s) {
    s->head = NULL;
}

void push(int x, stack *s) {
    node *q_push = (node *)malloc(sizeof(node)); //malloc is used to allocate memory for a node
    if (q_push == NULL) { //If it isn't possible to allocate memory. Malloc returns NULL
        printf("Unable to allocate memory.\n");
        return;
}
    q_push->data = x; //New node gets value of x
    q_push->next = s->head; //New node becomes new head
    s->head = q_push; //Updating the stack, so new node is the top
}
//*q_push is the pointer pointing to the newly allocated memory

int pop(stack *s) {
    if (empty(s)){
        printf("Cannot pop from an empty stack.\n");
        exit(EXIT_FAILURE);
    }

```

```c
    int q_pop = s->head->data; //Take the top value
    node *temp = s->head; //Make a place holder and make it equal to top node
    s->head = s->head->next; //Move the head to the next node
    free(temp); //Frees the memory of the popped node

    return q_pop;
}

bool empty(stack *s) {
    return s->head == NULL; //Returns true if head is NULL, which it is if empty
}

bool full(stack *s) {
    node *q_full = (node *)malloc(sizeof(node));
    if (q_full == NULL){
        //If malloc cannot allocate more memory it returns NULL
        return true;
    }
    free(q_full);
    return false; //When using linked lists, the stack is never technically full unless no more memory is available
}

```

(b) Test your implementation. **Create a new file**, where you include your stack library header file. You should expect the following "laws" to hold for any implementation of a stack. **Hint:** you should enforce these conditions using `assert` statements:

(1) After executing `initialize(s);` the stack s must be empty.

(2) After executing `push(x, s); y = pop(s);` the s must be the same as before execution of the two commands, and x must equal y.

(3) After executing `push(x0, s); push(x1, s); y0 = pop(s); y1 = pop(s);` the stack `s` must be the same as before execution of the four commands, `x0` must equal `y1`, and `x1` must equal `y0`.

```c
int main(){
stack s;

initialize(&s);
assert(empty(&s) == true); //After initializing the stack must be empty

push(10,&s);
assert(pop(&s) == 10);
//x = 10 and y = 10. The stack also looks the same as before the test.

push(20, &s);
push(30, &s);
assert(pop(&s) == 30);
assert(pop(&s) == 20);
//x0 = 20, x1 = 30, y0 = 30 and y = 20. The stack looks the same as before all four commands.
printf("All tests passed.\n");

return 0;

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
Out-0evn3iyi.szd' '--stderr=Microsoft-MIEngine-Error-xxcpnmmx.ptm' '--pid=Microsoft-MIEngine-Pid-azl52uxa.e0f
' '--interpreter=mi'
All tests passed.
```