PROGRAMMERING UGE 7

OPGAVE 1

Exercises

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).

HEADER FILE

Library.h | Library.c | Main.c

C: > Users > jonas > Desktop > AU > Programmering > Ny mappe > Library.h > taylor_sine(double, int)

```c
#ifndef library_H
#define library_H

double taylor_sine(double x, int n);

#endif
```

SOURCE FILE

Library.h | Library.c | Main.c

C: > Users > jonas > Desktop > AU > Programmering > Ny mappe > Library.c > ...

```c
#include <stdio.h>
#include <stdlib.h>

#include "library.h"



double taylor_sine(double x, int n) {

    double power = x;  // Gemmer power af x-værdien
    long factorial = 1;   // gemmer vores factorial
    int sign = -1;    // Det gør vi for at skifte fortegnet for hvert led
    double sum = x; // Summen er det første led

    for (int i = 1; i < n; i++) {
        power *= x * x;
        factorial *= (2 * i) * (2 * i + 1); // vores fac værdi
        sum += (sign * power / factorial);  // nye sum for hvert n
        sign = -sign;              // skifter fortegn
    }

    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.

MAIN FILE

Library.h | Library.c | Main.c

C: > Users > jonas > Desktop > AU > Programmering > Ny mappe > Main.c > main()

```c
#include <stdio.h>
#include "library.h"
#include <math.h>


int main(){
    double x = 1;
    int n = 1;

    double sum = taylor_sine(x, n);

    double ansi_result = sin(x);

    printf("Approksimationen af taylor series når x = %.2lf: %lf\n", x, sum);
    printf("ANSI sin(%.2lf) resulatat er: %lf\n", x, ansi_result);



    return 0;
}
```

Når x = 1 og n = 1

```
PS C:\Users\jonas\Desktop\AU\Programmering\Ny mappe>  gcc main.c library.c -o program
PS C:\Users\jonas\Desktop\AU\Programmering\Ny mappe> ./program
Approksimationen af taylor series nar x = 1.00: 1.000000
ANSI sin(1.00) resulatat er: 0.841471
```

- Her ser vi at approksimationen er ikke god nok når n = 1, fordi vores taylor funktion får værdien 1 når x = 1 og sin(1) = 0.841471.

Når x = 1 og n = 5

```
PS C:\Users\jonas\Desktop\AU\Programmering\Ny mappe>  gcc main.c library.c -o program
PS C:\Users\jonas\Desktop\AU\Programmering\Ny mappe> ./program
Approksimationen af taylor series nar x = 1.00: 0.841471
ANSI sin(1.00) resulatat er: 0.841471
```

- Når vi n bliver større bliver vores approksimation også tættere på det korrekte resultat, her får vi at taylor funktionen = sin(x), når n er 5.

OPAGVE 2

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.

Header filen:

stack.h > ...

```c
#ifndef STACK_H
#define STACK_H

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct Stack {
    Node* head;
} Stack;

void initialize(Stack* s);
void push(int x, Stack* s);
int pop(Stack* s);
bool empty(Stack* s);
bool full(Stack* s);

#endif
```

C filen med funktioner:

stack.c > ...

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

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

void push(int x, Stack* s) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = x;
    newNode->next = s->head;
    s->head = newNode;
}

bool empty(Stack* s){
    return s->head == NULL;
}

bool full(Stack* s) {
    return false; /*aldrig fuld*/
}

```

```c
int pop(Stack* s) {
    Node* temp = s->head;
    int popped = temp->data;
    s->head = s->head->next;
    free(temp);
    return popped;
}

```

(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`.

Vi har lavet en mainfunktion der tjekker om den overholder alle disse love med assert statements:

C stack_main.c > main(void)

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

int main(void) {
        Stack s;
    initialize(&s);

    // Test 1: Stack should be empty after initialization
    assert(empty(&s));

    // Test 2: Push and pop
    push(1, &s); // push (x, s)
    assert(empty(&s) == false);
    int y = pop(&s);
    assert(y == 1); // x must equal y
    assert(empty(&s)); // same as before the two commands

```

```
20
21      // Test 3:
22      push(2, &s); //x0
23      push(3, &s); //x1
24      int y0 = pop(&s);
25      int y1 = pop(&s);
26      assert(y0 == 3); //y0 = x1?
27      assert(y1 == 2); //y1 = x0?
28
29  printf("lortet virkede");
30  return 0;
31  }
32
```

Kører man den i terminalen som så kan man se at den ikke bliver stoppet af a assert statements men udføre printf:

```
PS C:\Users\Alex5\OneDrive\Dokumenter\AU 1. Semester\Programmering\Visual studio mappe> gcc stack_main.c stack.c -o program
PS C:\Users\Alex5\OneDrive\Dokumenter\AU 1. Semester\Programmering\Visual studio mappe> ./program
lortet virkede
```