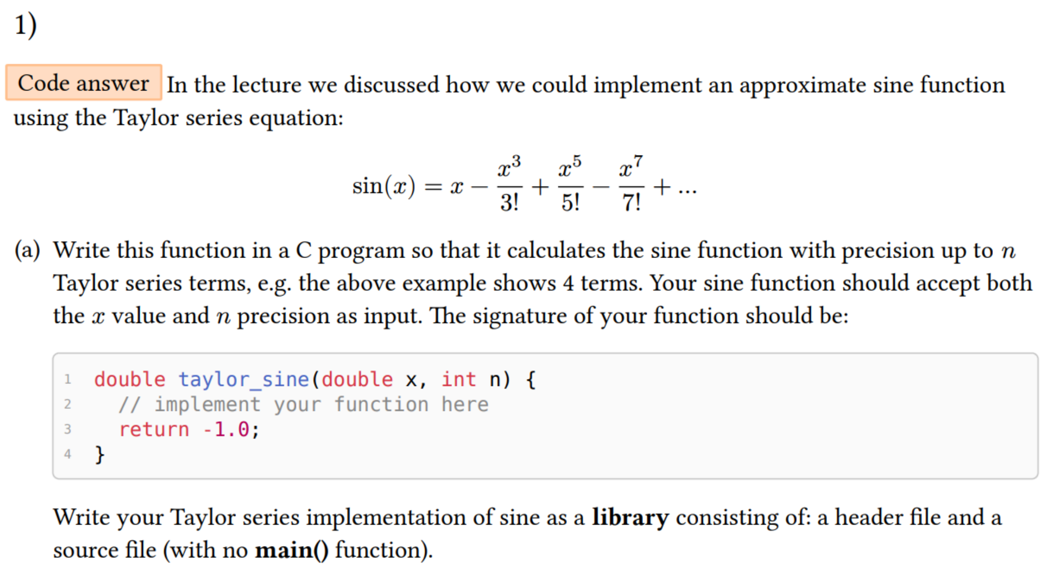
Programmering aflevering 7

Link til GitHub: [GitHub - IPFCE-2024/assignment-7-niya-og-amalie: ipfce-2024-classroom-assignment-7-285191U013-IPFCE-assignment-7 created by GitHub Classroom]()

Exercise 1



1)

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

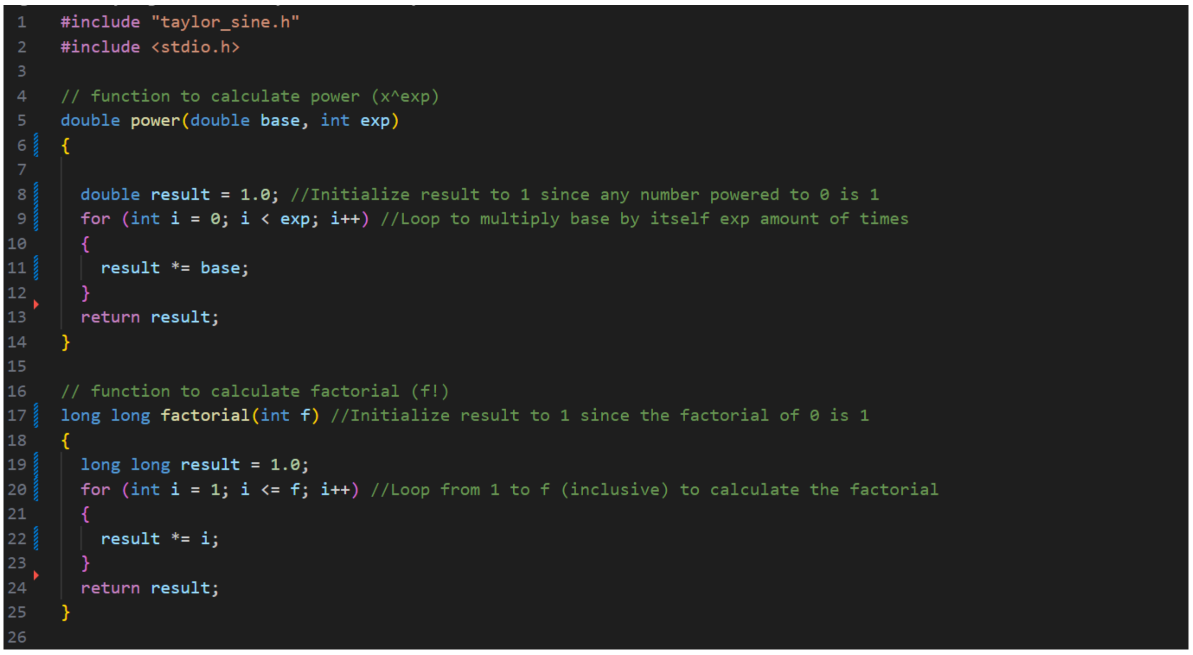
$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

(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:

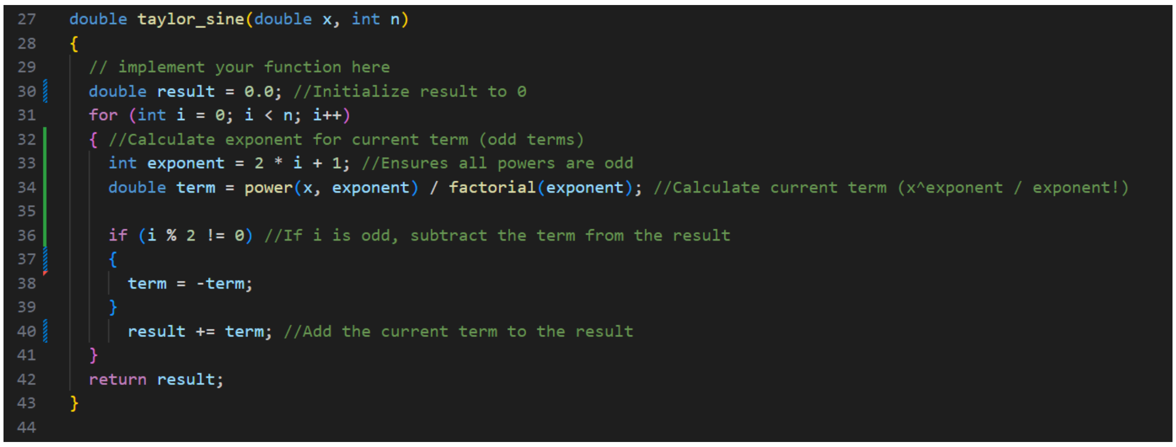
```c
double taylor_sine(double x, int n) {
  // implement your function here
  return -1.0;
}
```

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

c-fil:

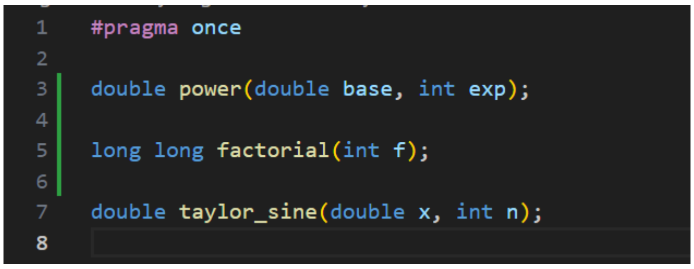


```c
#include "taylor_sine.h"
#include <stdio.h>

// function to calculate power (x^exp)
double power(double base, int exp)
{

  double result = 1.0; //Initialize result to 1 since any number powered to 0 is 1
  for (int i = 0; i < exp; i++) //Loop to multiply base by itself exp amount of times
  {
    result *= base;
  }
  return result;
}

// function to calculate factorial (f!)
long long factorial(int f) //Initialize result to 1 since the factorial of 0 is 1
{
  long long result = 1.0;
  for (int i = 1; i <= f; i++) //Loop from 1 to f (inclusive) to calculate the factorial
  {
    result *= i;
  }
  return result;
}

```



```c
double taylor_sine(double x, int n)
{
  // implement your function here
  double result = 0.0; //Initialize result to 0
  for (int i = 0; i < n; i++)
  { //Calculate exponent for current term (odd terms)
    int exponent = 2 * i + 1; //Ensures all powers are odd
    double term = power(x, exponent) / factorial(exponent); //Calculate current term (x^exponent / exponent!)

    if (i % 2 != 0) //If i is odd, subtract the term from the result
    {
      term = -term;
    }
      result += term; //Add the current term to the result
  }
  return result;
}
```

h-fil:



```c
#pragma once

double power(double base, int exp);

long long factorial(int f);

double taylor_sine(double x, int n);
```

(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.

Info

To use the ANSI C sin function do the following:

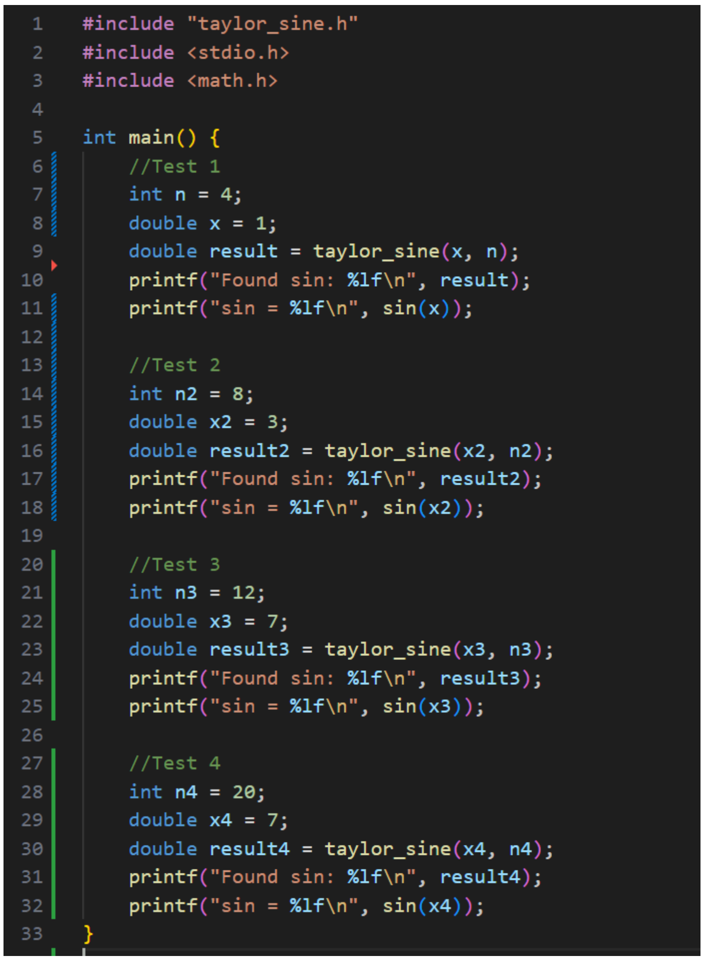
1. include the `math.h` header file.

```c
#include <math.h>

int main() {
  double pi = 3.14;
  printf("sin(pi) = %f\n", sin(pi));

  return 0;
}
```

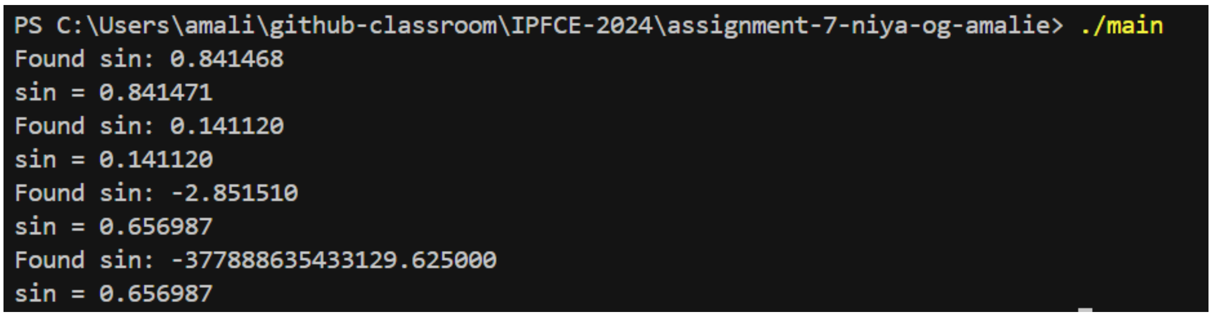
2. Link against `math.a`
   - If you are using `gcc`/`clang` in the terminal do `gcc <sourcefiles> -lm` or `clang <sourcefiles> -lm`
   - If you are using the "Run" button in VSCode see the guide from **assignment-4**.

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.

tests:



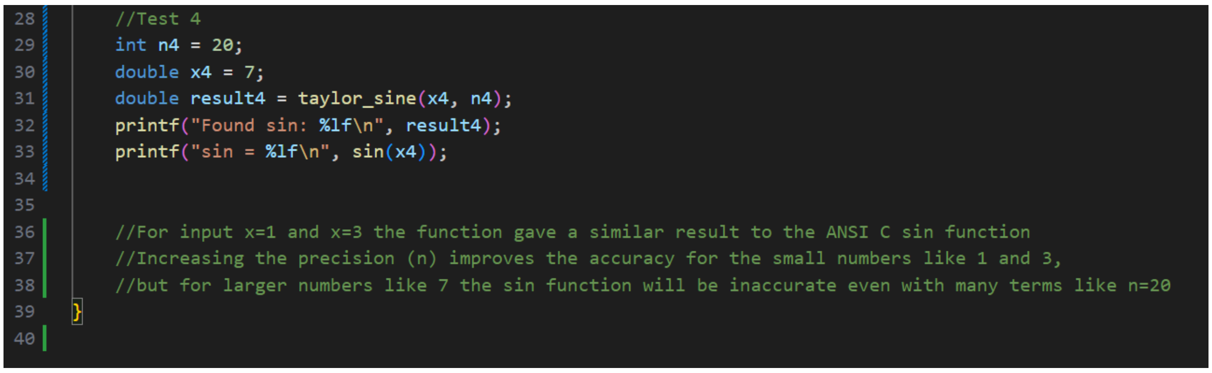
```c
#include "taylor_sine.h"
#include <stdio.h>
#include <math.h>

int main() {
    //Test 1
    int n = 4;
    double x = 1;
    double result = taylor_sine(x, n);
    printf("Found sin: %lf\n", result);
    printf("sin = %lf\n", sin(x));

    //Test 2
    int n2 = 8;
    double x2 = 3;
    double result2 = taylor_sine(x2, n2);
    printf("Found sin: %lf\n", result2);
    printf("sin = %lf\n", sin(x2));

    //Test 3
    int n3 = 12;
    double x3 = 7;
    double result3 = taylor_sine(x3, n3);
    printf("Found sin: %lf\n", result3);
    printf("sin = %lf\n", sin(x3));

    //Test 4
    int n4 = 20;
    double x4 = 7;
    double result4 = taylor_sine(x4, n4);
    printf("Found sin: %lf\n", result4);
    printf("sin = %lf\n", sin(x4));
}
```



```
PS C:\Users\amali\github-classroom\IPFCE-2024\assignment-7-niya-og-amalie> ./main
Found sin: 0.841468
sin = 0.841471
Found sin: 0.141120
sin = 0.141120
Found sin: -2.851510
sin = 0.656987
Found sin: -377888635433129.625000
sin = 0.656987
```

De to første tests har små afvigelser fra math.h’s værdier, mens test 3 og (særligt) test 4 afviger betydeligt.
Funktionen til beregning af sinus-værdier fungerer fint for små input, men bliver hurtigt upålidelig for større værdier.

(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)?



```c
    //Test 4
    int n4 = 20;
    double x4 = 7;
    double result4 = taylor_sine(x4, n4);
    printf("Found sin: %lf\n", result4);
    printf("sin = %lf\n", sin(x4));


    //For input x=1 and x=3 the function gave a similar result to the ANSI C sin function
    //Increasing the precision (n) improves the accuracy for the small numbers like 1 and 3,
    //but for larger numbers like 7 the sin function will be inaccurate even with many terms like n=20
}

```

Exercise 2

2)

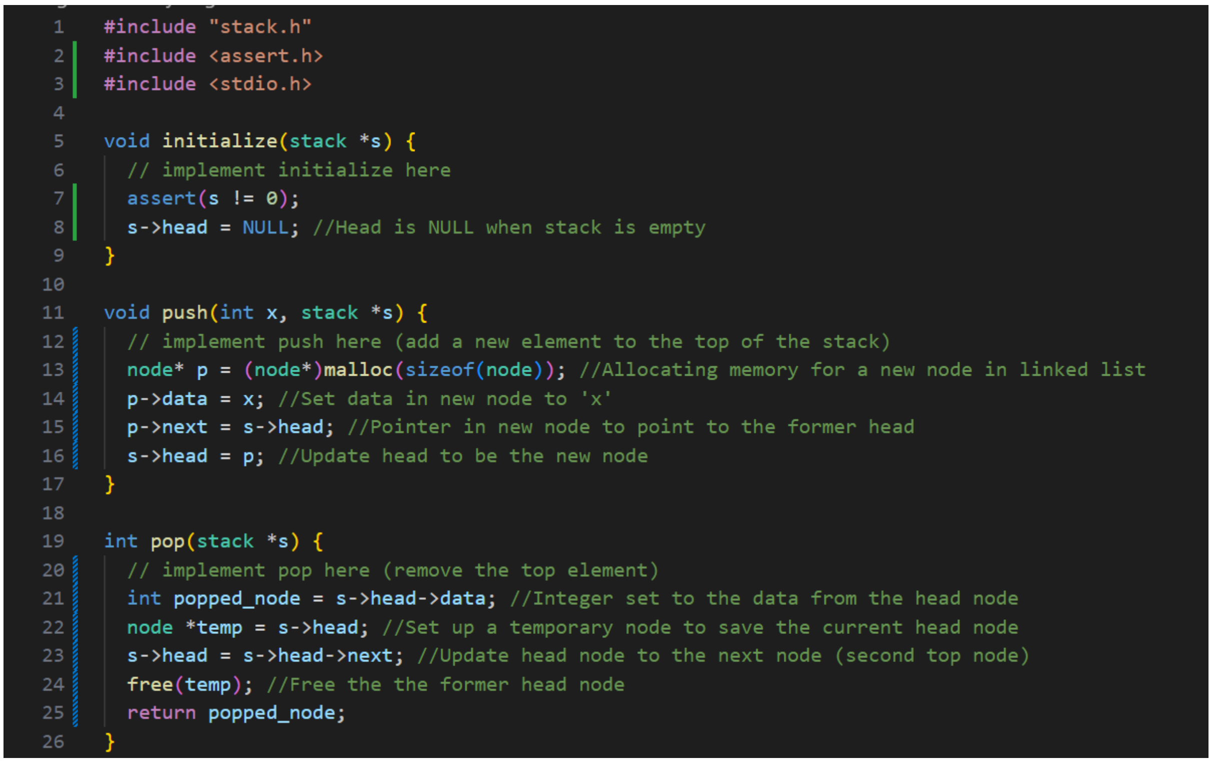
Code answer Stacks are containers where items are retrieved according to the order of insertion, independent of content. Stacks maintain *last-in, first-out order (LIFO)*. The abstract operations on a stack include:

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!*

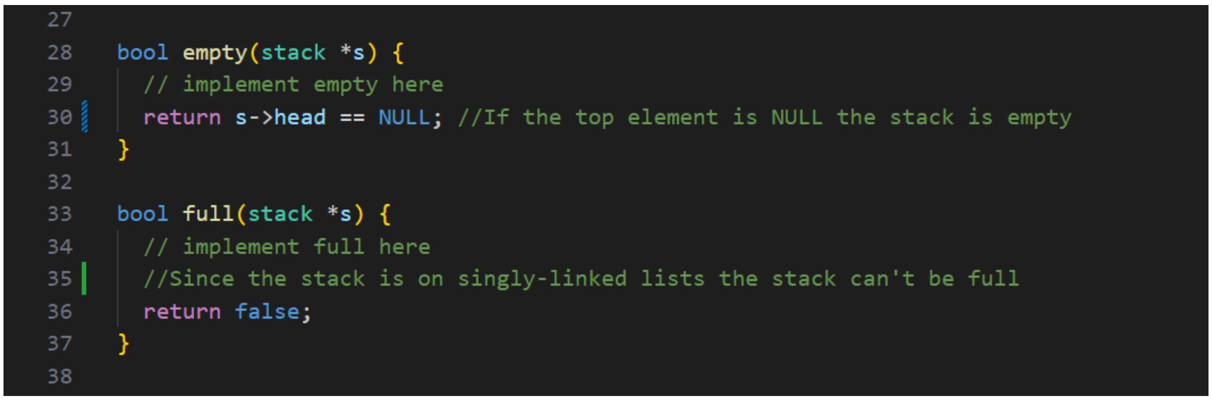
Note that there is no element search operation defined on standard stacks. Defining these abstract operations enables us to build a stack module to use and reuse without knowing the details of the implementation. The easiest implementation uses an array with an index variable to represent the top of 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.



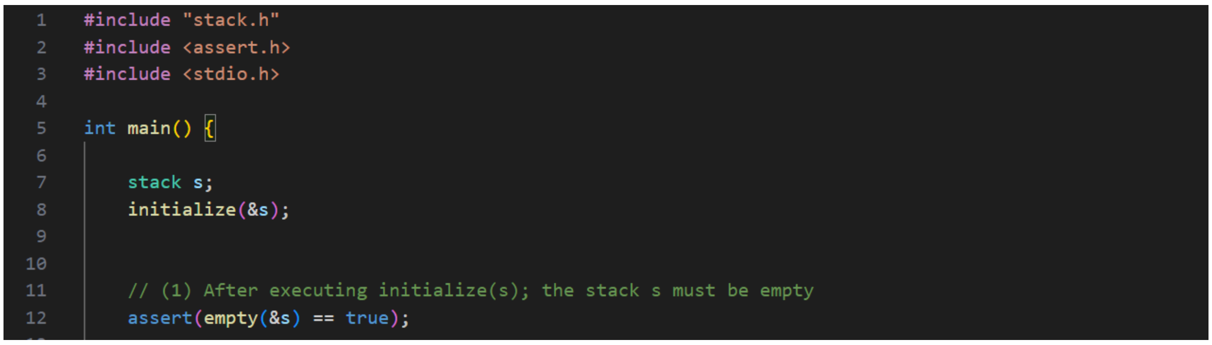
```c
#include "stack.h"
#include <assert.h>
#include <stdio.h>

void initialize(stack *s) {
  // implement initialize here
  assert(s != 0);
  s->head = NULL; //Head is NULL when stack is empty
}

void push(int x, stack *s) {
  // implement push here (add a new element to the top of the stack)
  node* p = (node*)malloc(sizeof(node)); //Allocating memory for a new node in linked list
  p->data = x; //Set data in new node to 'x'
  p->next = s->head; //Pointer in new node to point to the former head
  s->head = p; //Update head to be the new node
}

int pop(stack *s) {
  // implement pop here (remove the top element)
  int popped_node = s->head->data; //Integer set to the data from the head node
  node *temp = s->head; //Set up a temporary node to save the current head node
  s->head = s->head->next; //Update head node to the next node (second top node)
  free(temp); //Free the the former head node
  return popped_node;
}
```



```c

bool empty(stack *s) {
  // implement empty here
  return s->head == NULL; //If the top element is NULL the stack is empty
}

bool full(stack *s) {
  // implement full here
  //Since the stack is on singly-linked lists the stack can't be full
  return false;
}

```

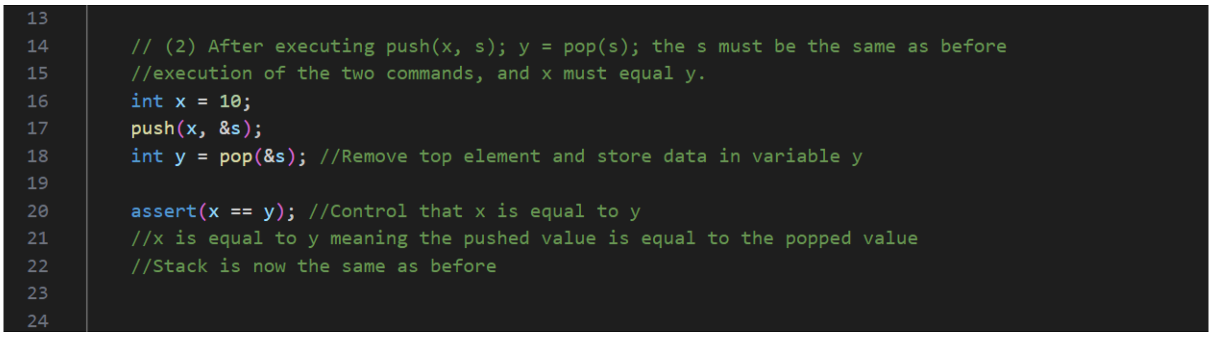
(b) Test your implementation. **Create a new file**, where you include your stack library header file. You should expect the following "laws" to hold for any implementation of a stack. **Hint:** you should enforce these conditions using `assert` statements:

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



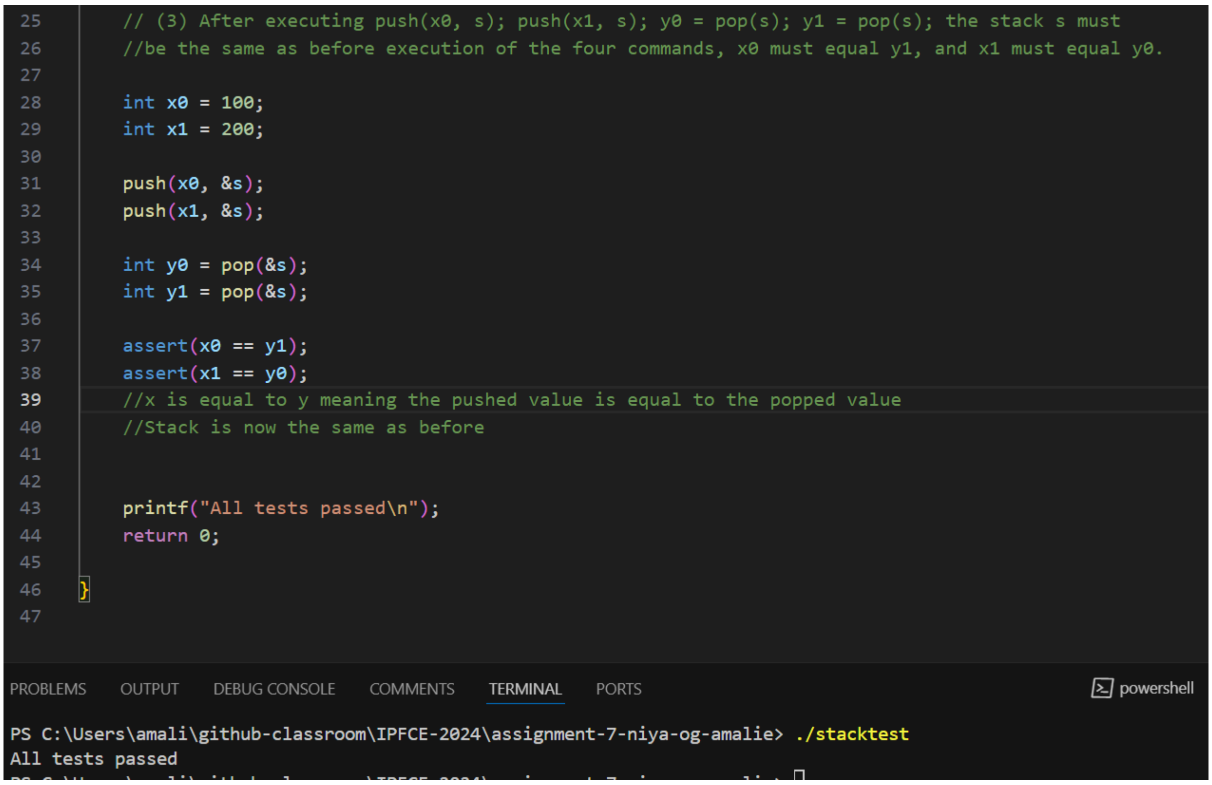
```c
#include "stack.h"
#include <assert.h>
#include <stdio.h>

int main() {

    stack s;
    initialize(&s);


    // (1) After executing initialize(s); the stack s must be empty
    assert(empty(&s) == true);
```

(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`.



```c

    // (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.
    int x = 10;
    push(x, &s);
    int y = pop(&s); //Remove top element and store data in variable y

    assert(x == y); //Control that x is equal to y
    //x is equal to y meaning the pushed value is equal to the popped value
    //Stack is now the same as before


```

(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
    // (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.

    int x0 = 100;
    int x1 = 200;

    push(x0, &s);
    push(x1, &s);

    int y0 = pop(&s);
    int y1 = pop(&s);

    assert(x0 == y1);
    assert(x1 == y0);
    //x is equal to y meaning the pushed value is equal to the popped value
    //Stack is now the same as before


    printf("All tests passed\n");
    return 0;

}
```

```
PS C:\Users\amali\github-classroom\IPFCE-2024\assignment-7-niya-og-amalie> ./stacktest
All tests passed
```