1 Exercise

Implement (type, don't copy-paste), compile and run the following program:

```
#include <stdio.h>
#include <stdlib.h>
#define N 1000


int main()
{
    int sum = 0;
    for (int i = 1; i <= N; ++i) {
        sum += i;
    }
    printf("%d\n", sum);

    return EXIT_SUCCESS;
}
```

2 Exercise

The Fibonacci sequence is given by

$$F_n = F_{n-1} + F_{n-2}$$

with $F_0 = 0$ and $F_1 = 1$.
Using the summation program as example, implement the Fibonacci sequence calculation

- Print each term (index and value) inside the loop
- First, print only the first 5 or 10 numbers to verify the calculation
- Next, print the first 50 Fibonacci numbers

3 Exercise

Back to the summation program

- Wrap the summation in a loop itself, and reset sum inside the outer loop, so it runs longer
- Find the number of iterations (inner times outer) where the program runs just a few seconds, without optimizations. You can use the bash / Linux function time: `time <program>`(Using time is not precise, but a good approximation for a few seconds)
- Now compile with -O3 (full optimization), and do the same
- Now use a non-trivial summation, e.g.: `sum += log(i);`

  Use -O3, `#include <math.h>` and link with libm (-lm flag)
- Now remove the final printf function, and do the same

4 Exercise

- Put the Fibonacci calculation inside a function.Call it from main, and print its return value there
- Add functionality to return -1 as an error value when the input argument is incorrect (negative or too large)
- Extra: make a recursive Fibonacci function (remove the for-loop). Note: functions calls are expensive: a for-loop is much faster than a series of (recursive) function calls

5 Exercise

- Move the Fibonacci function into its own file (or move the main function into its own file): `fibo.c` and `main.c`
- Create a fibo.h file that has the declaration of the function. Include it in fibo.c and main.c
- Compile `fibo.c` and `main.c` to objects file (`fibo.o` and `main.o`): `gcc -c <other flags> fibo.c`
  Link the object files together into an executable
- Turn the `fibo.o` object file into a (static) library `fibo.a`
- Compile and link `main.c` with the `fibo.a` in one go
- Extra: create a Makefile to do all the work

6 Exercise

- Define a struct Ball that contains an x and y position, and an vx and vy velocity
- Instantiate the struct at a global level (outside main and other functions).Pick some decent initial values;
- Create a function that takes as inputs
  - a time interval tdelta
  - a gravity acceleration constant g
  - and a damping factor f

  and calculates the next position & velocity of the ball. The ball bounces when `y <= 0` , and loses vy by a factor f (as well as reverses vy).
- In main(), set tdelta, g and f as variables (not constants)
- Set a variable stop, which is the total time of the simulation
- Start at t = 0, and loop until t > stop
- Use printf to print the time, x and y position each loop iteration
- You can redirect the output to a file: `./ball > ballpos.txt`
- You can create a plot of, for example, height versus time in any spreadsheet
- Or if you like to try gnuplot: `gnuplot> plot ballpos.txt using 1:3 with lines title ball`

7 Exercise

Take the file with the ball main function, and add the following before the loop. scanf reads input from the command line (and requires conversion specifiers like printf), and we can ask a user for input. Note that scanf cant print output (e.g., a question string). scanf waits for <enter> before fully reading the input, and discards whitespace

```
int main()
{
    double g = 9.8, f = 0, tdelta = 1, stop = 1;
    printf( Value for g:    );
    scanf("%lf", &g);
    printf( Value for f:    );
    scanf("%lf", &f);

    return 0;
}
```

`"\%lf"` indicates a double explicitly. Think long float.
NB: scanf returns the number of correctly read variables. Use it to verify everything went ok.

8 Exercise

Move the global ball variable into the main function.
Change the function that calculates the balls movement to accept a pointer to a Ball struct (as well as the usual g, f, tdelta arguments).
Dont forget to change the call to the function in main as well.

9 Exercise

Store the results of the ball calculation in three arrays: timestamps, xpos and ypos. You have to calculate the array size from the input time parameters, and dynamically allocate the array (and free the allocation afterwards). To convert a double to an int, you cast it:

```
  int a = (int) 5.0;
```

Be aware that this truncates the result (that is, rounds down). For example:

```
int a = (int) (10.0/3.0);
a == 3
```

Move the printf function out the main loop, and create a second loop afterwards where you only print the results. (Normally, the calculations would all be done in one function, and printing of the results in another function.)

10 Exercise

Change the ball program and compile it with a C++ compiler. Youll run into a few incompatibilities:
Cast the return value of malloc in C++:

```
double *timestamps = (double *) malloc(n * sizeof *timestamps);
```

While you wont get a warning doing this in C, dont do it: it hides a potential error (forgotten `#include <stdlib.h>`).
This is why C++ != C with some extras
C++ does not have designated initializers for structs. Instead, use

```
struct Ball ball = {0, 10, 1, 0};
```

You can do this as well in C, but it is less clear than using the member names. In C++, structs are quite different beasts, and initialization like this is not really used.

11 Exercise

Replace malloc/free with std::vector

12 Exercise

Replace printf/scanf with std::cout/std::cin

13 Exercise

Replace the Ball struct pointer with a reference

14 Exercise

Implement the ball calculation inside the Ball struct

- use class instead of struct
- your choice of private / public
- your choice of member variable naming style
- gravity g and efficiency f should become member variables as well

For the constructor: gravity and efficiency should be mandatory argumentsThe position and velocity variables should be default arguments.

Extra

- Store the positions and time in a vector, as member variables of the class
- Add a member function that takes care of the full loop: it should take a start and stop time, and a timestep. It should call the calculation function each iteration.

  Thus, everything we did in main(), is now done inside the class (apart from the output)
- Since youre now not calling the calculation directly, you could make it private.
- Youll need to make the relevant output variables public, or define a getter function, if you want to be able to read and print the final results

  Or: you could define a print() or output() class member function for this.