# CS Basics - Exercises
# Control Structures in C

C. Grothoff and E. Benoist

Fall Term 2017-18

## 1 Library

Make sure you have `libtool` installed for this exercise. You will also need to add `LT_INIT` to your `configure.ac` file to create C libraries.

Write two files "myLib.c" and "myLib.h" that will contain a library. You will use that library in a file myProgram.c.

- The file `myLib.c` contains the definitions of the functions logarithm(long) and exponential(long). You may use the example of last week.
- The file `myLib.h` declares the functions (prototypes).
- The file `myProgram.c` uses all the functions.

In order to work together, you must use a "make" process for compiling and linking all the files.

You may need to use the number `EULER` equals to 2.718281...

### 1.1 New functions

Expand your library with the following functions:

- `long square(long)`: that computes the square of a `long`
- `long gcd(long, long)` that computes the greater common divisor of two longs

## 2 Arrays

- Write a function for initializing an array (given as parameter) to a value given as a parameter. The input parameters are: the array, its size, the new value cells will contain.

  Example of use:

```
long myArray[256];
initArray(myArray,256,0);
```

- Write a function that takes as input a `long`, an array of `longs` (that will be defined in the `main`) and the length of the array. You will write inside the array the prime factors of your parameter. Terminate the list of prime factors with a 0. Your function should return 0 on success and 1 if the array was too short.

  Example of use:

  ```
  int failed = primeFactors(123,myArray,arrayLength);
  ```

- Write a function that displays all the elements of an array, until one is 0. Use the function to show the prime factors of a given number.

  Example of use

  ```
  printarray(myArray);
  ```

- Write a function for computing the *standard deviation* for all the elements in an array. You first need to include the function `average()` seen during the course.

- Move all your new functions inside your library (.c and .h files). Compile your library in a .o and link it with a main program.

## 3 Fast prime time

Implement a program to quickly count the number of primes below a given number. The number must be read from `stdin`. You should use the Sieve of Eratosthenes and multiplications instead of division.

Why can you not use a global variable for the array?

Use `gdb` to understand the execution of your program.

```
$ gcc -O0 -g prog.c -o binary
$ gdb binary
(gdb) break main
(gdb) CTRL-x a
(gdb) run
(gdb) print EXPRESSION
(gdb) s
(gdb) n
```

## 4 Faster prime time

Keep a version of your original program around.

Optimize your program to use an array of `uint64_t` storing 64 bits. Use bit shift operations (`<<`) to address the bits. Use functions to manipulate the main array (this will require passing the array as an argument).

What is the difference between 1, 1L, 1LL and 1ULL, and why does it matter for your code?

## 5 Fastest prime time

Keep a version of your previous program around.

Optimize your program by not reserving space for even numbers in the sieve.

Optimize your program by not counting bits individually in the end, and instead use GCC's `__builtin_popcountll()`.

## 6 Measure

Compile your code with *optimizations* enabled:

```
$ gcc -O2 prog.c -o binary
```

Then run it with the `time` command to observe the runtime:

```
$ echo 400000000 | time ./binary
```

Compare different optimization levels (O0, O1, O2, O3) for your different implementations.

## 7 Bonus task

Use `valgrind`'s cachegrind tool and `kcachegrind` to analyze the caching behavior of your code. Can you optimize it further?

## 8 Bonus task

Read up on bit counting optimization using bit masks and arrays as common CS hiring questions.