# CMPUT 350 Lab 1 Exercise Problems

---

Rules:

- You can use all course material and man pages, but no other information such as web pages, books, or written notes. Using other information sources during the exercise constitues cheating.
- Your programs must compile without warning using

  ```
  g++ -g -Wall -Wextra -Wconversion -Wsign-conversion -O -std=c++17 ...
  ```

  In case there are compiler warnings or errors, marks will be deducted.
- Test your programs with different values. For now, the speed of your program is irrelevant. So don't spend time on optimization
- You must check for the appropriate preconditions/postconditions. Your program shouldn't crash or have undefined behaviour (**hint**: use asserts)!
- Your programs must be well structured and documented. Use ctrl-x t in Emacs to pretty-print it. Marks are assigned to functionality, program appearance, and comments.
- In case your program hangs, use ctrl-c to terminate it.
- Remember that you need to include the appropriate header files. To find out which ones you need for specific functions such as printf, use then man command.

Submit your solution files `p1.cpp p2.cpp` on eClass under "Lab 1 / Submission".

**Important**: Submit often (the system will only accept solutions submitted before 16:50)

---

1. [9 marks] Write C++ program `p1.cpp` which implements and tests function

   ```
   void rotate_right(int A[], int n);
   ```

`rotate_right` shifts all elements of n-element array `A` one location to the right, and stores the original right-most entry into the left-most position.

**Examples**:

- `A[]` before call: 1 2 3 4 5 6
- `A[]` after call: 6 1 2 3 4 5
- after another call: 5 6 1 2 3 4
- etc.

Test your function using a couple of test cases in `main()` including one which rotates an n-element array `n` times and checks whether the resulting array is the same as the one you started with.

---

2. [12 marks] Write C++ program `p2.cpp` that reads a sequence of int array pairs (in decimal notation with white-space characters [space, newline, tab] as delimiter) from `stdin` and prints their component-wise product to stdout.

Inputs consist of multiple instances. Your program must handle them all and only stop when encountering the end of the input or an input error. You may assume that each individual instance fits in memory. In case of an input error, your program **needs** to report "`input error`" to `stderr` and **return with value 1**. If everything is OK, the program needs to **return with value 0**.

**Example**: Suppose `input.txt` contains the following 3 instances:

```
10
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
5
2 3 4 -1 6
1 1 1 1 1
1
8
5
```

Running the command `./p2 < input.txt` should then yield the following output:

```
10
0 8 14 18 20 20 18 14 8 0
5
2 3 4 -1 6
1
40
```

In general, the input looks as follows:

```
Instance_1
Instance_2
...
Instance_k
```

for which each instance is described by:

```
n            # (> 0) length of the two input arrays
n ints       # first array
n ints       # second array
```

The output for each instance has the form:

```
n            # length of output array
n ints       # output array (= product of two input arrays)
```

Handle each input instance one at a time, and outputting the result before moving onto the next instance.

**Hints**:

- All you need for input is `std::cin >> input`; where input is an int variable (the line format is irrelevant, what matters is that all numbers are separated by white-space characters, which `cin >> input` consumes but ignores)
- For testing, prepare text files with various inputs (legal and illegal) and redirect them into your application like so: `./a.out < test-input`
- Use `valgrind` to check for memory leaks (see Lab 0)