

# CSE333 Exercise 5

**Out:** Monday, January 24

**Due:** Friday, January 28 by 11 am PST

**Rating:** 1 (note)

## Goals

- Write a C++ program from scratch
- Write code that reads in user input from stdin, and parses the input
- Utilize streams in C++
- Use the C++ linter

## Background

As discussed in lecture, C++ and C have a lot of similarities, yet can be different in a lot of ways. In this exercise, you will write your first C++ program from scratch that, like Exercise 1, parses user input to perform some arithmetic computation. We want you to compare your experience with this exercise and Exercise 1 to see some of the similarities and differences between C and C++.

## Problem Description

Write a C++ program that:

- Prompts the user for a positive integer ( $> 0$ , strictly greater than 0).
- Prints out all integers that are **factors** of that integer; feel free to use the simplest, brute-force factorization algorithm you can think of. You should only print each factor once, in ascending order.

For example, when we run our exercise and the user inputs the number 100, the interaction with the terminal should look like:

```
bash$ ./ex5
Which positive integer would you like me to factorize? 100
1 2 4 5 10 20 25 50 100
bash$
```

- On the second line, there is a space between the prompt and the user input ('100').
- On the third line (*i.e.*, the factors), there is no space before the '1' or after the '100'.

## Implementation Notes

### Input/Output Streams

You should use the `iostream` library to read user input from the terminal and to write to the terminal. Specifically, you should investigate using `std::cin`, `std::cout`, and `std::cerr`. You can find more information by looking at `iostream` in the C++ reference (<http://www.cplusplus.com/reference/iostream/>).

### User Input

Since you will be reading in user input from `stdin`, you should make sure your code handles various inputs from the user, which may be in an unexpected format. For this exercise, you should investigate to see how `std::cin` parses input and checks for errors, and then make sure your code utilizes this error checking for bad input. Once you detect a bad input, please print out usefull error message and exit from the program. Do *NOT* try to loop and prompt the user again.

## Style Focus

Don't forget that the good practices from previous exercises still apply, however, some guidelines from C are *superseded* by C++ guidelines!

### C++ Style

In general, write C++-style code when possible. For example, although both `printf` and `std::cout` can achieve the task of writing to the terminal, you should use `std::cout` since this is a C++ program.

### cpplint

Since we are now dealing with C++ instead of C, it is appropriate that we use a linter designed for C++. There are three ways to get `cpplint.py`:

1. Direct download from this link ([../hw/cpplint.py](https://courses.cs.washington.edu/courses/cse333/22wi/hw/cpplint.py)).
2. From your terminal, run:

```
bash$ wget https://courses.cs.washington.edu/courses/cse333/22wi/hw/cpplint.py
```

3. It will be included in your Gitlab repo when hw2 is released.

Be sure to fix any warnings that this new linter reports! You may have to run `chmod +x cpplint.py` to make `cpplint.py` executable.

### Using-Directives

Make sure that if you use any `using` directives, that you list out each individual object/function you use, and not an entire namespace. In other words, do NOT use `using namespace std;`, instead list out everything you are using from that namespace with lines like: `using std::cout;` .

## Submission


You will submit: `ex5.cc` .

Your code must:

- Compile without errors or warnings on CSE Linux machines (lab workstations, `attu` , or CSE home VM).
- Have no runtime errors, memory leaks, or memory errors ( `g++` and `valgrind` ).
- Be contained in the file listed above that compiles with the command:

```
bash$ g++ -Wall -g -std=c++17 -o ex5 ex5.cc
```

- Have a comment at the top of your `.cc` file with your name(s) and CSE or UW email address(es).
- Be pretty: the formatting, modularization, variable and function names, commenting, and so on should be consistent with class style guidelines. Additionally, the linter shouldn't have any complaints about your code ( `cpplint.py` ).
- Be robust: your code should deal with hard-to-handle/edge cases and bogus user input (if there are any) gracefully.

Submit your code on  Gradescope (<https://www.gradescope.com>). Don't forget to add your partner if you have one.