ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE, Spring Semester 2022
Lecturer: Dr. R. Käppeli, Prof. O. Schenk
Assistants: T. Holt, P. Bansal, M. Lechekhab,
D. Pasadakis, M. Averseng, X. Niu, Z. Easa
R. Worreby, N. Siegenheim

HPC Primer in C++

This project will introduce you to basic programming concepts for HPC. It is meant to harmonize the basic programming skills of incoming students.

1. Compiling and running a program on the Euler cluster

The file `read_environment.cpp` is a simple C++ program that reads some environment variables and prints it to the console. Your first task is to log onto the Euler cluster , compile, and then run this program. To log onto the Euler cluster you will need to use the Secure Shell (SSH) cryptographic network protocol. This allows you to open up a terminal on a remote machine. You may need to be logged into the VPN if you are not connected to the ETHZ network.

```
[user@localhost]$ ssh userid@euler.ethz.ch
[user@eu-login]$
```

Once you successfully log into the cluster you will be connected to the login node. The login node manages basic administration of the cluster and is the common interface for all users. **You should only compile and run small programs on the login node for testing and debugging purposes.** You will know that you are connected to the login node if the hostname in your cursor contains the word "login"; for example, in the code snippet shown above the hostname is `eu-login`. **Programs with large execution times or with large memory requirements should be run on the compute nodes using the batch system.** To learn more about how to use the batch system, see `https://scicomp.ethz.ch/wiki/Using_the_batch_system`.
To compile your program, you must load the GNU compiler:

```
[user@eu-login]$ module load gcc
```

NB: You will need to re-load your required modules each time you connect to a node.
You can see the modules that are currently loaded using the command:

```
[user@eu-login]$ module list
```

Next you need to compile the program. To do this you will use the GNU C++ compiler `g++` (if we were compiling C code we would use `gcc`).

```
[user@eu-login]$ g++ -o helloEuler read_environment.cpp
```

What we have done with this command is to run the program "`g++`" and we have passed it three arguments separated by space:

1. The option flag `-o` which notifies the program that the next argument will contain the desired name of the compiled output file.
2. The output filename that we wish to specify.
3. The name of the file that we wish to compile.

You should now have a binary file named `helloEuler` in your directory. Execute this binary, passing it your first and last name in camelCase as a command line argument.
To run on the login node:

```
[user@eu-login]$ ./helloEuler firstnameLastname
```

To run on a compute node interactively, use:

```
[user@eu-login]$ bsub -I ./helloEuler firstnameLastname
```

else use:

```
[user@eu-login]$ bsub ./helloEuler firstnameLastname
```

and read the output log file `lsf.xxxx`.

2. Implementing a basic fraction toolbox in C++

In the `fraction_summing` directory you will find the skeleton of a program to do some basic operations with fractions. Header files help you organize your codebase more effectively by moving function and variable declarations to a file seperate from the main program. In the header file `fraction_toolbox.hpp` you will find a number of function declarations. In the C/C++ programming language function declarations are generally of the form:

```
functionReturnDataType functionName(inputDataType inputName);
```

In the directory you will also find the source file `fraction_toolbox.cpp` where the functions are actually defined. Please complete the following tasks in the fraction toolbox files in order to implement a basic fraction toolbox (each item will have a TODO comment in the source code):

1. In the header file define a C datatype `struct fraction` to store a rational number (i.e., a fraction of integers). The numerator and the denominator of the fraction should be named `num` and `denom` respectively.
2. Write a function `square_fraction()` that takes a `fraction` structure as input and returns its square as a `fraction` while leaving the original input unchanged.
3. Write a function, `square_fraction_inplace()`, that takes a `fraction` structure as input and squares that `fraction` without returning any data. Add the function declaration to `fraction_toolbox.hpp`.
4. Write a function, `fraction2double()`, that takes a `fraction` as input and returns its value as a double precision floating point number.
5. Write a function, `gcd()`, that uses the *recursive* Euclid's algorithm (see Algorithm 1 below) to return the greatest common divisor of two integers.
6. Write a function, `gcd()` (overloading the `gcd()` function call), that takes a `fraction` as input and returns the greatest common divisor (integer) of the numerator and denominator using the *iterative* Euclid's algorithm (see Algorithm 2 below).
7. Write a function, `reduce_fraction_inplace()`, to reduce a fraction in place using either of the `gcd()` functions. Add a short comment to your source code that states which of the two `gcd()` functions your code is using (state the line numbers), and how you know it is using that one. Add the function declaration to `fraction_toolbox.hpp`.
8. Write a function, `add_fractions()`, that takes two `fraction` structures as input, and returns their sum as a new `fraction` in it's most reduced form.
9. Write a function, `sum_fraction_array_approx()`, that takes a pointer to an array of fractions and `n` (the length of the array) as input, and returns a double precision floating point number approximately equal to the sum of the array. Why is this an approximate function?
10. Write a function, `sum_fraction_array()`, that takes a pointer to an array of fractions and `n` (the length of the array) as input, and returns a `fraction` equal to the sum of the array, by summing the `fraction` objects.

Euclid's Algorithm

Euclid's Algorithm, discovered by the Greek mathemetician Euclid in 300BC, is a method for finding the greatest common divisor of two integers. The algorithm can be implemented either recursively or iteratively, below we provide the pseudocode for both versions. **NB:** The $\leftarrow$ symbol represents the assignment operator. In most programming languages this operator is represented by the $=$ symbol, although mathematically speaking this choice of symbols is questionable.

Algorithm 1: Euclidean Algorithm - Recursive

Data: Two integers a and b
Result: Integer greatest common divisor of a and b
Function `gcd`(a, b)**:**
  if $b = 0$ **then**
    return a
  else
    return `gcd`$(b,\ a \mod b)$
  end

Algorithm 2: Euclidean Algorithm - Iterative

Data: Two integers a and b
Result: Integer greatest common divisor of a and b
Function `gcd`(a, b)**:**
  while $b \neq 0$ **do**
    $t \leftarrow b$
    $b \leftarrow a \mod b$
    $a \leftarrow t$
  end
return a

3. Testing the Function Toolbox

The `main.cpp` file is where you will find the skeleton of the testing program for your fraction toolbox. All of the functions that you declared in the header file will be available in your main program due to the include statement:

```
#include "fraction_toolbox.hpp"
```

Since compiling this program requires managing dependencies across different files that must be compiled, we use a makefile to automate compiling. To compile code using a makefile simply use:

```
[user@eu-login]$ make
```

You may also clean out all compiled binaries by using:

```
[user@eu-login]$ make clean
```

The `main.cpp` file also contains the `main()` function. In C code, this is the only function that will be executed when the code is run, thus if you want other functions to be executed when you run the code, you must call them in the `main()` function. Keep this in mind as you are building and testing the various other parts of this assignment. Please implement the following functions in the `main.cpp` file to test your toolbox:

1. Write a function, `test23467()`, that reads command line arguments (you may use the pre-implemented `readcmdline()` function) to create a fraction, then tests functions 2, 3, 4, 6, and 7 by calling the functions and printing the results. Please also print descriptions of all of the results so that it is obvious which results represent which function. There is a pre-implemented print function for fractions, you can write your own print statements for testing functions 4 and 6.

2. Write a function, `test5()`, that prompts the user for two integers and then prints the greatest common divisor of those two integers. **Hint:** check out the `cin` object.

3. Write a function, `test_array_functions()`, that:
   a) takes integer n as an input;
   b) dynamically allocates a memory buffer for an array of fractions of length n using the malloc function;
   c) passes a pointer to that memory buffer to the pre-implemented function `fill_fraction_array()`. The buffer will then be filled with an array of fractions of the form $\left[\frac{1}{1(1+1)}, \frac{1}{2(1+2)}, \cdots, \frac{1}{n(1+n)}\right]$. **Hint:** see pre-implemented `print_fraction_array()` function to verify results;
   d) passes a pointer to the (now filled) memory buffer to your `sum_fraction_array()` function;
   e) prints resultant sum;
   f) passes a pointer to the memory buffer to your `sum_fraction_array_approx()` function;
   g) prints the resultant sum.

   Test this function in the `main()` function and ensure that it is giving you the results you would expect for small n. At some large n the `sum_fraction_array()` function will start giving funny results. Find that n, and then write a brief explaination of what is happening as a comment in the source code. Why is this not happening to the approximate function? **Hint:** the values of 32-bit integers are between -2^{31} and $+2^{31}$.

4. Once you have finished implementing the testing statements please call the `test_toolbox()` function from the `main()` function.