Lab 1

Intro

My name is Jarek Nowisz. I am a PhD student. The best way to reach me is to write at: j.nowisz@gmail.com.

There is no better way to learn than to practice. Before each lab I will try to prepare for you a file with the task(s) and hints to help you solve the problem. You will work on your own, but I expect you to connect to the lab teams meetings. During our meetings I am waiting for your questions and I will help you solve problems you enqounter.

Rules

1. You will be given tasks to solve.
2. You will submit working software with solutions to me for evaluation.
3. All your programs should compile and run on Windows or on Linux. Sorry: I don't have access to other platforms.
4. Every solution should have a README.txt file with description how to compile and run the program.

Task for today: Task 1.

Together with this file you received an `example_in_c.c` file. It is written in `C` and your task is to rewrite it into `C++`.

Let's start with some quick analysis of what the program does.

It defines a structure of a complex number:

```
typedef struct _CMPLX
{
    double Re, Im;
} CMPLX;
```

where `Re` is real part and `Im` is imaginary part of the number.

We have also 4 functions defined:

1. `ReadC` - reads 2 parts of the complex number form the user's input.
2. `AddC` - adds two complex numbers and returns a new complex number which is their sum.
3. `AbsC` - calculates the absolute value of a complex number.
4. `AddCA` - adds complex numbers from two tables and modifies values of the third table filling them with the results of sum operations.

At the end we have a `main` function that uses the 4 functions defined earlier. The program expects the user to input 2 complex numbers (4 numbers, 2 for each complex number). Then adds these numbers and calculates and prints the absolute value of the sum. It also adds 3 pairs of predefined complex numbers from 2 arrays, stores the sums in the third array and then prints the sums.

Lets check if the `C` program compiles and works:

On Linux run in terminal:

```
gcc example_in_c.c -lm ; ./a.out
```

The `-lm` param is a link to the math library in which sqrt function used in the program is implemented.

On Windows: 1. start Visual Studio 2. create an empty c++ console project 3. add file `example_in_c.c` 4. modify file `example_in_c.c` by changing `scanf` to `scanf_s` twice in the `ReadC` 5. compile and run

You can play with the program:

- Enter first complex number (C1): 1 1
- Enter second complex number (C2): 2 3
- Your first complex number is (C1): 1+1i
- Your second complex number is (C2): 2+3i
- Sum of C1 + C2: 3+4i
- Abs of C1 + C2: 5
- First array of complex numbers is (A1): 3+4i 1+3i 5-2i
- Second array of complex numbers is (A2): 3+4i 1+3i 5-2i
- Sum of A1 + A2: 6+8i 2+6i 10-4i

Create the simplest, empty program in C++

Create a file `main.cpp` and put there an empty `main()` function. Remember that in `C++` the `main` function must return int. It may optionally have `argc` and `argv` parameters. More info here. A file with the `main` function does not have to have the name `main`. You can use any name, but it is a good practice to use `main.cpp` (or with any other extension, see: next chapter) to quickly find the main function definition. Don't forget to return some value from main. It is a good practice to return 0 on success and some other values indicating errors.

Compile, run it and see if it works. The program should start and finish without any output printed to the console.

On Linux run in terminal:

```
g++ main.cpp; ./a.out
```

On Windows: 0. close solution 1. create new solution 2. create an empty c++ console project 3. create and fill `main.cpp` file 4. compile and run

Split the program into separate files

For small, simple programs all the code can be written in a single file. It is a good practice to split the code into separate files. Our example is small and simple, but still we are going to split it into 3 files: 1. a header `cmplx.h` file with the declaration of the Cmplx structure, 2. an implementation `cmplx.cpp` file with the declaration of the Cmplx structure, 3. an `main.cpp` file with the `main` function that was already created.

The extensions of the files with `C++` code can be anything, but it is good to use extensions, that are meaningful for others. People usually use: .H, .h, .hpp extensions for headre files (or no extensions al all), and .C, .cc, .cpp, .CPP, .c++, .cp, or .cxx for implementation files. I personally prefer and use .hpp and .cxx.

In implementation files (.cxx) files you should put definitions of functions (methods). In header files (.hpp) you usually put declarations of functions (methods) and definitions of classes.

1. create empty files: `cmplx.h`, `cmplx.cpp`
2. insert into `cmplx.h`:

```
#ifndef __CMPLX_H__
#define __CMPLX_H__

    //your code goes here

#endif
```

The code above is called include guards and it will help you avoid including code from an included header file multiple times and compiler errors. You may instead use `#pragma once` directive. 3. add `#include` statements into `cmplx.cpp` and `main.cpp` files. At the top of both files add line:

```
#include "cmplx.h"
```

Again: compile, run it and see if it works. The program should start and finish without any output printed to the console.

On Linux run in terminal:

```
g++ cmplx.h cmplx.cpp main.cpp; ./a.out
```

On Windows: recompile and run

Add declaration of our complex type

1. In `cmplx.h` file instead of `//your code goes here` type:

```
struct Cmplx
{
    double real_part;
    double imaginary_part;
};
```

Don't forget about semicolon at the end of the class or struct definition. It is mandatory. Structs and classes don't differ much in `C++`: the only difference is in visibility of internals. You will learn about it later during the lecture. For now we will use a struct.

2. in `main.cpp`, insode `main` function declare a variable of type `Cmplx`:

```
Cmplx a;
```

Again: compile, run it and see if it works. The program should start and finish without any output printed to the console.

On Linux run in terminal:

```
g++ cmplx.h cmplx.cpp main.cpp; ./a.out
```

On Windows: recompile and run

Declare and define a method `print` for our complex type

In `C` and in our example (example_in_c.c) we printed the values of complex number by accessing `Im` and `Re` internals of the struct. Every part of the program in `C` can access and modify values of internals of structs or other complex types. It is very flexible solution but has a few drawbacks. The main are: 1. the logic that operates on the internals of the struct may be spread across the application 2. it is more difficult to controll who, when and why changes or accesses the internal data In C++ we address above problems by introducing encapsulation and methods that operate only on data of the type for which they are declared.

Let's implement a `print` method that prints the value of our complex number. Instead of declaring a function we declare a method of our `Cmplx` struct. We can do it in two ways: 1. define a method inside the declaration of the `Cmplx` struct in `cmplx.h` file, or 2. declare a method inside the declaration of the `Cmplx` struct in `cmplx.h` file and define it in the `cmplx.cpp` file.

Whenever possible choose the second approach. We will use now the second approach too: 1. add a declaration of the `print` method in the `Cmplx` struct in `cmplx.h` file:

```
struct Cmplx
{
    double real_part;
    double imaginary_part;
    void print();
};
```

2. add the declaration of the `print` method in the `cmplx.cpp` file:

```
#include "cmplx.h"
#include <iostream>

void Cmplx::print()
{
    std::cout << this->real_part << "+" << this->imaginary_part << "i" << std::endl;
}
```

3. print the value of our "empty" complex number in `main` function in the `main.cpp` file:

```
#include "cmplx.h"

int main()
{
    Cmplx a;
    a.print();
    return 0;
}
```

Method `print` was declared inside the struct `Cmplx`. Notice that we prepend the definition of the function in `cmplx.cpp` file with a struct (or class) name: `Cmplx::print()` to distinguish it from definitions of `print()` methods of other classes or structs. To use it, we have to call it like this: `object.print()` or `pointer_to_object->print()`, for example we called it in our `main` function like this: `a.print()`. The `print` method has access to all internals of objects of the `Cmplx` type, so it can print `real_part` and `imaginary_part`. In the body of the `print` method we can access them like: `this->real_part` or just `real_part`.

More about the body of the `print` method: `cout` is an output stream it is declared in `std` namespace, so we use it like: `std::cout`. It has an operator `<<` which writes into the stream, so you can write for example:

```
std::cout << 5;
```

or

```
std::cout << "dog";
```

The `std::cout << std::endl;` writes a new line into the stream. The `<<` operator not only writes into the stream, but also returns the modified sstream so we can chain more `<<` operators. Instead of this:

```
std::cout << this->real_part;
std::cout << "+";
std::cout << this->imaginary_part;
std::cout << "i"
std::cout << std::endl;
```

we can write:

```
std::cout << this->real_part << "+" << this->imaginary_part << "i" << std::endl;
```

we can even simplify the code more by removing `this`:

```
std::cout << real_part << "+" << imaginary_part << "i" << std::endl;
```

Again: compile, run it and see if it works. This time the program should write some numbers to the console in the forat: A+Bi, where A and B are numbers. The values of A and B are random because valueas of `real_part` and `imaginary_part` were not initialized. We will deal with it later.

On Linux run in terminal:

```
g++ cmplx.h cmplx.cpp main.cpp; ./a.out
```

On Windows: compile and run

Add initialization to our complex numbers

As you could see in the previous section the values of the complex number that were printed were not initialized. We will add a constructor to set the initial values when declaring objects of Cmplx `struct`: 1. Inside the `Cmplx` struct, in the file 'cmplx.h` add:

```
Cmplx(double real_value, double imaginary_value);
```

2. In file `cmplx.cpp` add definition:

```
Cmplx::Cmplx(double real_value, double imaginary_value)
    : real_part(real_value), imaginary_part(imaginary_value)
{
}
```

We will not go into details of the constructor. For now it is enough to know, that when constructing an object the values of parameters will be used to initialize internal parameters of the just created object. 3. In the `main` function change:

```
Cmplx a;
```

to:

```
Cmplx a(3, 4);
```

Compile, run it and see if it works. This time the program should write exactly: `3+4i`.

On Linux run in terminal:

```
g++ cmplx.h cmplx.cpp main.cpp; ./a.out
```

On Windows: compile and run

Create add method to count sums of complex numbers

Just like the `print` method we now create an `add` method. It is called on one complex number and accepts the second complex number as an argument. Notice, that we can implement it in two variants: 1. It may modify internal values of the calling object: internal values are increased by values from the complex number passed in parameter, or 2. It may use the internal values of both complex numbers, add them and return a third complex number as a sum.

This is the implementation of the second variant: 1. in file `cmplx.h`, inside `struct Cmplx`, add line:

```
Cmplx add(Cmplx &other);
```

2. in file `cmplx.cpp` add definition:

```
Cmplx Cmplx::add(Cmplx &other)
{
    Cmplx result(this->real_part + other.real_part, this->imaginary_part + other.imaginary_part);
    return result;
}
```

3. add some code in `main` function to test the new `add` method:

```
Cmplx a(3, 4);
Cmplx b(5, 6);
Cmplx c = a.add(b);
a.print();
b.print();
c.print();
```

Compile, run it and see if it works.

On Linux run in terminal:

```
g++ cmplx.h cmplx.cpp main.cpp; ./a.out
```

On Windows: compile and run

This time the program should write exactly:

```
3+4i
5+6i
8+10i
```

Create `abs` method

Hint 1: check here definition of the absolute value for complex numbers Hint 2: the `abs` method should return `double` and does not need any parameters. Hint 3: the code should be similar to `add` and `print` methods.

Create `read` method

Hint 1: read the documentation about `cin` and `>>` operator. Hint 2: the `read` method should be very similar to the `print` method. Hint 3: don't forget to print some info for the user before accepting input from her/him, for example: 'Enter real part of the complex number:' or 'Enter imaginary part of the complex number:'

Declare and sort the table with complex numbers by its absolute value

Hint 1: you can adapt and use the sorting algorith from `sorting_in_cpp.cpp` . Hint 2: you can alternatively use the standard `sort` method; read about about it here. Hint 3: to declare an array of 3 complex numbers: `1+2i` , `3+4i` , and `5+6i` you can write: `Cmplx numbers[] = {{1, 2}, {3, 4}, {5, 6}};`