Exercise in C++ Programming for CE
ExC++PCE
Winter Term 2014/2015

Mini Project (Complex Numbers)

General Remarks

StudOn submission

This mini project is an individual project, i.e., each student has to work by or herself. *No group-work*! Please note that, as for the StudOn submissions of the questions on the assignment sheets, the successful completion of this mini project is mandatory. Please check the StudOn course[1] regularly for further information about the deadline and the submission of you solutions.

Complex Numbers

A complex number $z \in \mathbb{C}$ consists of two parts, a real part (x) and an imaginary part (y). It is defined as $z = x + iy$ in Cartesian notation, where $i^2 = -1$. Complex numbers can also be represented in polar notation, as $z = r(cos\varphi + isin\varphi)$, or in Euler notation, as $z = re^{i\varphi}$. All representations can be converted into each other, via trigonometric functions. See Figure 1 for an illustration of a complex number z on the two-dimensional Cartesian plane.

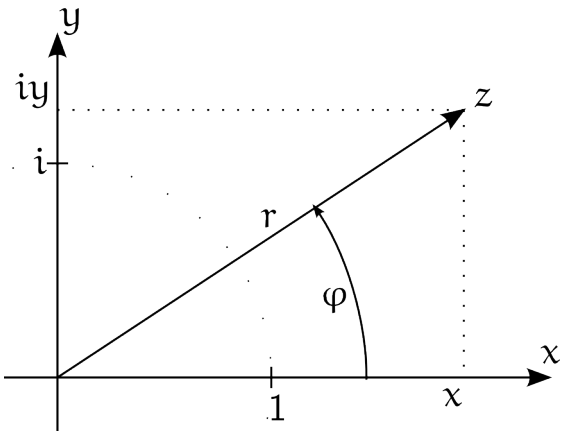

Figure 1: Complex number z on Cartesian grid.

Computations on complex numbers comprise the same basic operations as "real-valued" arithmetic, namely addition, subtraction, multiplication, division, exponentiation and the computation of roots. Additionally, complex numbers require/permit a unique operation, namely the computation of the complex conjugate, which, for a complex number $z = x + iy$ is defined as $\bar{z} = x - iy$, i.e., a reflection on the abscissa.

[1] `http://www.studon.uni-erlangen.de/studon/goto.php?target=crs_1061841`

Tasks

1 Class Complex

Develop a C++ class for complex numbers. The class shall provide *all necessary members*, in terms of *variables*, *methods*, *operators* and (helper-)functions to perform complex number arithmetic. This includes, yet is not restricted to, addition, subtraction, multiplication and division of complex numbers, as well as the aforementioned conversions between the different notations. Also, the complex conjugate, the n-th power and the n-th root of any complex number shall be computable.

Consider looking at std::complex[2] as an "inspiration" for your own complex number class.

It is sufficient for your class and implementation to use and operate on double precision values.

2 Compiling & Testing of Class Complex

Write a Makefile that can be used to

1. compile your class (to an object file) (*make complex*)
2. compile your test driver (*make test*)
3. generate the documentation (*make doc*)
4. do everything (*make all*)
5. clean the directory from intermediate files (*make clean*)

Write a test driver for your test (i.e., a class TestComplex), and verify it with enough sensible test cases, to be sure of the "correctness" of your complex number class and its operations.

Your test driver shall read input from the command line and output results to the command line as well. We will feed it a list of test cases in prefix notation, see the following table for an example:

<operation/operator>	<operand 1>	<operand 2>	<output>
(+)	(3 1.5)	(1.0 2)	(4 3.5)
(sub)	(2 -5)	(-3.3 4)	(5.3 -9)
(==)	(1 2)	(2 1)	(F)

We expect the output format to match the input format, i.e., numbers are in parenthesis, separated by one blank space between the two numbers and a dot as decimal separator. The first number is always the real part and the second number the imaginary part of the complex, both of them may be prefixed with a '-' for negative numbers, the '+' is always omitted. Boolean return values have to be a capital letter; T for true and F for false. The operation (sub, add, div, mul)/operator (-,+,/,*) starts at the beginning of the line and is separated from the first operand by one blank character.

We will provide a small sample test file for you to verify your implementation.

[2] http://en.cppreference.com/w/cpp/numeric/complex

3 Documentation

Annotate your code using Doxygen[3] in order to produce useful documentation for your implementation.

1. Generate configuration: doxygen -g Doxyfile
2. Edit configuration: [vim/nano/emacs/...] Doxyfile
3. Modify configuration to match your project structure (set INPUT and OUPUT_DIRECTORY accordingly)
4. Annotate your code
5. Generate documentation: doxygen Doxyfile

For a very brief introduction a few small examples see:

`http://www.yolinux.com/TUTORIALS/LinuxTutorialC++CodingStyle.html#DOXYGEN`

More Remarks

We do not require you to use a particular coding style, but consider and try to stick to a consistent coding standard throughout the project. If you are interested in Bjarne Stroustrup's opinion on coding standards and some further reading on coding styles at Google, check out the following websites:

`http://www.stroustrup.com/bs_faq2.html#coding-standard`.

`http://google-styleguide.googlecode.com/svn/trunk/cppguide.html`

The submission of you implementation must be in line with the following directory structure. This structure is very common to organize larger projects in order to keep source, header and documentation well separated.

```
<matriculation#>_proj1/
├── src/
│   └── cpp files
├── include/
│   └── header files
├── test/
│   ├── cpp file(s) for test driver
│   └── test driver for your solution
├── doc/
│   ├── Doxyfile
│   ├── Documentation
│   └── [Doxygen stuff]
└── Makefile
```

The submission of the project will be via the StudOn course website. Please check regularly for updates and comments on the assignment and the format of the submission.

[3] `http://www.doxygen.org`