Linköping university
ITN

TNG033: Programming in C++
Lab 2

Course goals

To use

- base-classes and derived classes,
- class inheritance,
- polymorphism, dynamic binding, and virtual functions,
- abstract classes,
- and, operator overloading: `operator+=`, `operator+`, `operator[]`, and `operator()`.

Preparation

You must perform the tasks listed below before the start of the lab session *Lab2 HA*. In each *HA* lab session, your lab assistant has the possibility to discuss about three questions with each group. Thus, it is important that you read the lab descriptions and do the indicated preparation steps so that you can make the best use of your time during the lab sessions.

- Review the notion of operator overloading. This concept was introduced in lecture 7.
- Do the set 2 of exercises. Pay special attention to exercise 4.
- Do the set 3 of exercises. Pay special attention to exercise 4.
- Review the concepts introduced in lectures 9 and 10, e.g. inheritance, virtual functions, abstract classes.
- Do the set 4 of exercises. Pay special attention to exercise 4.
- Download the files for this exercise. Then, you can use CMake to create a project with the downloaded files.
- Write the interface of class `Expression` whose specification is given below. Implement this class.
- Write the interface of class `Polynomial` (i.e. `Polynomial.h)`, according to the given specification.
- Finally, implement class `Polynomial`. Thus, before the start of lab session *Lab2 HA*, your code should pass the test phases 0 to 9 of the test program given in `main.cpp`. As for lab1, assertions are used to test the code.
- Write down the most important questions that you want to discuss with your lab assistant.

If you have any specific question about the exercises, then send us an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in English or Swedish. Add the course code to the e-mail's subject, i.e. "`TNG033/MT2: ...`".

Exercise

The aim of this exercise is that you define a (simplified) polymorphic class hierarchy to represent certain types of expressions, as described below.

Class `Expression`

Define a class **`Expression`** to represent mathematical functions of the form $y = f(x)$, i.e. functions of one real argument x. This class should offer the following basic functionality.

- A function, called `isRoot`, to test whether a given value x is a root of the function f.
- An overloaded function call operator (`operator()`) to evaluate an expression `E`, given a value `d` for variable x, i.e. `E(d)` returns the value of expression `E` when x gets the value `d`.
- A stream insertion operator `operator<<` to write an expression to a given output stream.
- All expressions should be clonable. A class is clonable if its instances can create copies ("*clones*") of themselves. Thus, for any instance `o` of class `Expression`, `o.clone()` should return a pointer to a copy of object `o`.

Class `Polynomial`

Define a subclass **`Polynomial`** that represents a polynomial, *e.g.* $2.2 + 4.4x + 2x^2 + 5x^3$. Your class should provide the following functionality, in addition to the basic functionality for an expression.

- A constructor that creates a polynomial from a vector of `double`s. For example, consider the following array declaration.

  ```
  std::vector<double> v{2.2, 4.4, 2.0};
  ```

  Then,

  ```
  Polynomial q{v};
  ```

  creates the polynomial $q = 2.2 + 4.4x + 2x^2$.
- A conversion constructor to convert a real constant into a polynomial.
- A copy constructor.
- An assignment operator.
- An add-and-assign operator (i.e. `operator+=`).
- Addition of two polynomials `p+q`, where `p` and `q` are polynomials.
- Addition of a polynomial with a real (`double`) value d, i.e. `p+`d and d`+p`, where `p` is a polynomial. Expressions such as `p +=` d`;` should also compile.
- A subscript operator, `operator[]`, that can be used to access the value of a polynomial's coefficient. For instance, if `p` is the polynomial $2.2 + 4.4x + 2x^2 + 5x^3$ then `p[3]` should return `5`. Note that statements such as

  `k = p[i];` or

  `p[i] = k; //modify the coefficient of x`^{i}

should both compile with the expected behaviour, where `p` is a polynomial and `k` is a variable.

The polynomial's coefficients can be saved either in a C-array or in a vector ([`std::vector`]). You take this implementation decision. You may need to add other member functions like destructors, though any added member functions should not extend or restrict the described functionality for the class.

When writing a polynomial to an output stream, the coefficients should be written with two digits after the decimal point and the zero coefficients should also be written. Follow the examples given in the tests in `main.cpp`. An example is given below.

```
p = 3.40 + 0.00 * X^1 + 5.00 * X^2 + 5.00 * X^3
```

A subscript operator was discussed in lecture 7 for class `Matrix`, `Matrix::operator()`. Recall that to access a slot of a `Matrix` object two indices are needed, line and column. Thus for the example of class `Matrix`, the `operator()` was overload instead of `operator[]` because the latter can only have one argument, while the former can have any number of arguments.

The test phases 0 to 9 of the test program given in `main.cpp` test the class `Polynomial`. Note that there is no guarantee that your code is correct just because it passes all these tests. You can add extra tests to the file `main.cpp`. However, the tests you add must be clearly indicated in the code. You cannot remove any of the given tests, though.

Class `Logarithm`

Define another subclass (of `Expression`) named `Logarithm` that represents a logarithmic function of the form $c1 + c2 \times log_b(E)$, where E is an expression (either a polynomial or a logarithm) and $c1$, $c2$, and b are constants. In this exercise, you can assume that $b > 1$. You can find below some examples of logarithmic expressions that should be representable.

- $6 + 3.3 \times log_2 x$ or $2.2 \times log_2(x^2 - 1)$ or $-1 + 3 \times log_{10}(log_2(x^2 - 1) + 2.2)$.

`Logarithm` class should provide all the functionality described for expressions. Moreover, this class should also provide the following operations.

- A default constructor that creates the logarithm $log_2 x$.
- A constructor that given an expression E, and constants $c1$, $c2$, and b creates the logarithmic expression $c1 + c2 \times log_b(E)$.
- A copy constructor.
- An assignment operator.
- A function named `set_base` that modifies the base of the logarithm to a given integer b' (you can assume that $b' > 1$).

You may need to add other member functions like destructors, though any added member functions should not extend or restrict the described functionality for the class.

When writing a logarithm to an output stream, the constants $c1$ and $c2$ should be written with two digits after the decimal point and they should always be written, even if their value is zero. Follow the examples given in the tests in `main.cpp`. An example is given below.

```
l1 = 0.00 + 1.00 * Log_2( 0.00 + 1.00 * X^1 )
```

The test phases 10 to 15 given in `main.cpp` test the class `Logarithm`. Note that there is no guarantee that your code is correct just because it passes all these tests. You can add extra tests to the file `main.cpp`. However, the tests you add must be clearly indicated in the code. You cannot remove any of the given tests, though.

Code requirements

The member function `isRoot` requires to compare the result of the evaluation of an expression with zero. Since in this lab we are dealing with floating point arithmetic, the computations may not be precise. Consequently, it is possible to obtain a value different from zero, though an expression evaluation should be mathematically equivalent to zero.

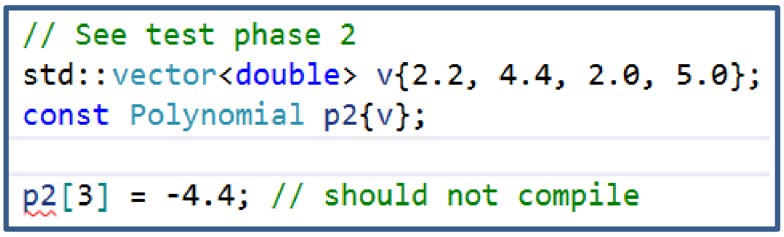
This problem was discussed in the TND012[1] course (see the notes "Digital storage of integers and floating points"). There is no universal solution for this problem, i.e. the solution usually depends on the problem and the variables' meaning. For this lab, we suggest the following to test whether the value of `E(d)` is equal zero, for an expression `E`.

```
if ( std::abs(E(d)) < EPSILON ) …
```

where `EPSILON` is a small constant like

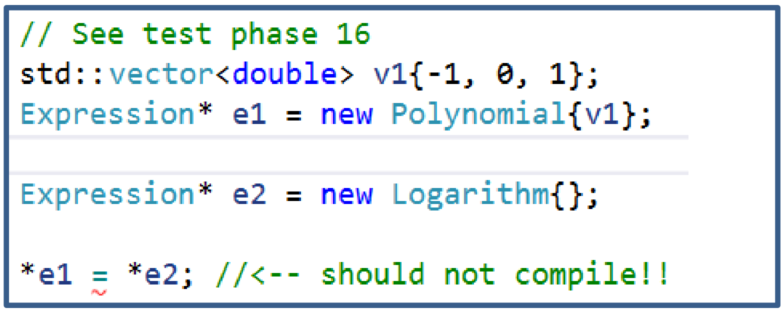
```
constexpr double EPSILON = 1.0e-5;
```

Make also sure that code similar to the following examples does not compile and reflect on the reason for not wanting such pieces of code to compile.



```
// See test phase 2
std::vector<double> v{2.2, 4.4, 2.0, 5.0};
const Polynomial p2{v};

p2[3] = -4.4; // should not compile
```



```
// See test phase 16
std::vector<double> v1{-1, 0, 1};
Expression* e1 = new Polynomial{v1};

Expression* e2 = new Logarithm{};

*e1 = *e2; //<-- should not compile!!
```

Theory questions

While presenting your lab, you need to answer the following two questions, though other questions can also be asked. We expect that you reflect on these questions in advance.

- Investigate the reason(s) for having the member function `clone`. If the idea is to make copies of objects that are instances of class `Expression` then aren't the copy constructors in each of the derived classes of `Expression` enough?
- How can one prevent compilation of the pieces of code referred in "*Code requirements*"?

[1] Use login: **`TND012`** and password: **`TND012ht2_12`**.

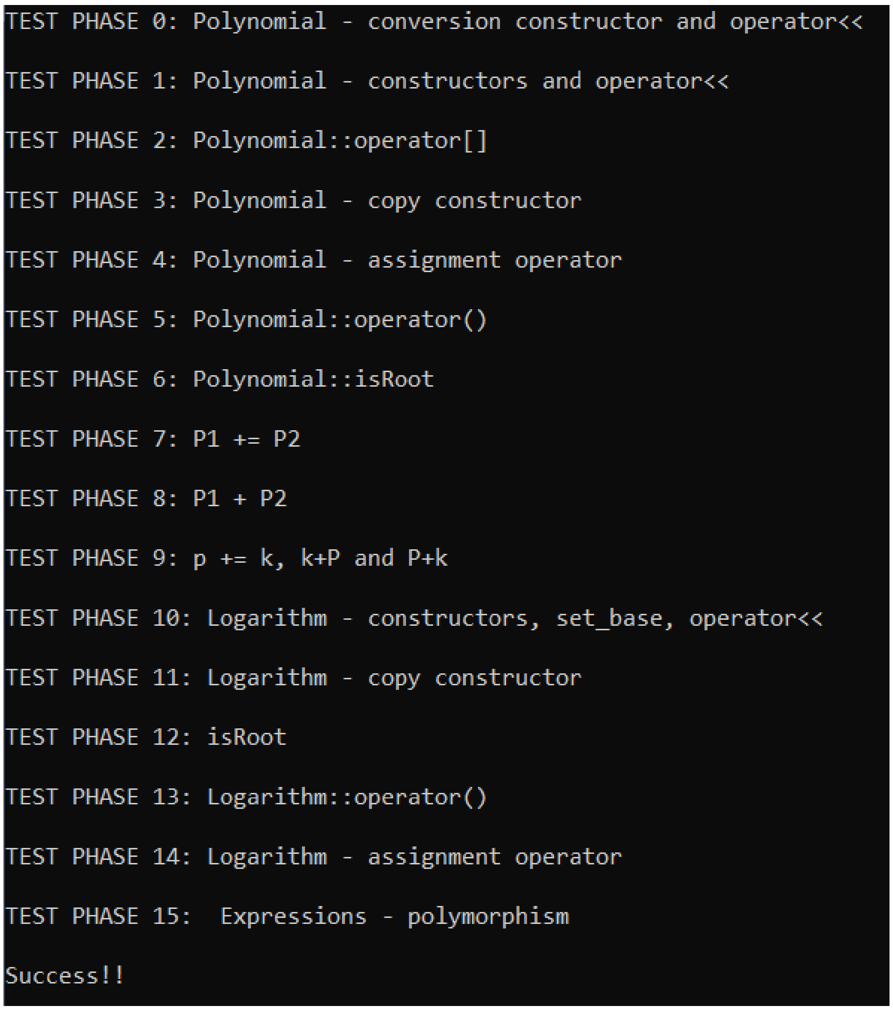
Presenting solutions and deadline

The exercises in this lab are compulsory and you should demonstrate your solutions during the lab session *Lab2 RE*. Read the instructions given in the labs web page and consult the course schedule. We also remind you that your code for the lab exercises cannot be sent by email to the staff.

Necessary requirements for approving your lab are given below.

- Use of global variables is not allowed, but global constants are accepted.
- The code must be readable, well-indented, and follow good programming practices.
- The compiler must not issue warnings when compiling your code.
- There are no memory leaks neither other other memory-related programming errors.
- Your code must pass all tests in the given file `main.cpp`.
- Answer the theory questions.

The given test program (file `main.cpp`) output is given below.



```
TEST PHASE 0: Polynomial - conversion constructor and operator<<

TEST PHASE 1: Polynomial - constructors and operator<<

TEST PHASE 2: Polynomial::operator[]

TEST PHASE 3: Polynomial - copy constructor

TEST PHASE 4: Polynomial - assignment operator

TEST PHASE 5: Polynomial::operator()

TEST PHASE 6: Polynomial::isRoot

TEST PHASE 7: P1 += P2

TEST PHASE 8: P1 + P2

TEST PHASE 9: p += k, k+P and P+k

TEST PHASE 10: Logarithm - constructors, set_base, operator<<

TEST PHASE 11: Logarithm - copy constructor

TEST PHASE 12: isRoot

TEST PHASE 13: Logarithm::operator()

TEST PHASE 14: Logarithm - assignment operator

TEST PHASE 15:  Expressions - polymorphism

Success!!
```

Lycka till!