Linköping university
ITN

TNG033: Programming in C++
Lab 2

Course goals

To use

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

Preparation

You must perform the tasks listed below before the start of the lab session *Lab2 HA*.

- Review the notion of operator overloading. This concept was introduced in lecture 6 and lecture 7.
- Review the concepts introduced in lecture 9 and lecture 10, e.g. inheritance, virtual functions, abstract classes.
- Do exercises 4 and 5, of set 4 of exercises.
- Download the files for this exercise. You can then use CMake to create a project with the downloaded files.
- Add the definition of class `Expression` (to `expression.h`), according to the specification given below. Implement the member functions of this class.
- Add the definition of class `Polynomial` (to `polynomial.h`), according to the given specification.
- Finally, implement the member functions of 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 `test.cpp`. As for lab1, assertions are used to test the code.

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: ...`".

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 variable. 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.

- A type conversion operator supporting conversion from expressions to `std::string`, though implicit conversions should not be supported.
- 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.
- A function `clone` supporting polymorphic copying. 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 vector.

  ```
  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.
- When converting `Polynomial` to `std::string`, coefficients should be represented with two digits after the decimal point and zero coefficients should also be represented. Follow the examples given in the tests in `test.cpp`. An example is given below.

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

The polynomial's coefficients should be saved in a vector (`std::vector`). Check whether your class also needs a destructor.

Recall that the concept of overloading the subscript operator was discussed in lecture 7.

The test phases 0 to 9 of the test program given in `test.cpp` test the class `Polynomial`. Note that in the `main` there are also commented tests that should not compile (search for "`should not compile`"). Make sure you also perform these tests (just uncomment those lines and check that the program cannot compile) and pay attention to the compilation error issued.

There is no guarantee that your code is correct just because it passes all the tests. You can add extra tests to the `main` function. 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 real constants. In this exercise, you can assume that b is an integer larger than one. 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 $b > 1$ 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 $b' > 1$).
- When converting from `Logarithm` to `std::string`, constants $c1$ and $c2$ should be represented with two digits after the decimal point. Note that $c1$ and $c2$ should always appear in the string, even if their value is zero. Follow the examples given in `test.cpp`. An example is given below.

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

Check whether your class also needs a destructor.

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

Requirements

- Polynomials' coefficients should be saved in a vector (`std::vector`).
- You can add extra tests to the `main` function. However, the tests you add must be clearly indicated in the code. You cannot remove any of the given tests, though.

- 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" and exercise 5 of övning1). 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;
```

- Code similar to the following examples should not compile. Reflect also 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 and bring an answer that you can motivate.

- 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?
- Motivate why is desirable that the two pieces of code in "*Requirements*" do not compile?
- How does your solution prevent the compilation of the pieces of code mentioned above?
- Explain the concept of object slicing and give an example. How does your code prevent object slicing?

[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.

- The code must be readable, well-indented, and follow good programming practices.
- The compiler must not issue warnings.
- There are no memory leaks neither other other memory-related programming errors.
- Your code must pass all tests in `test.cpp`.
- The code must satisfy the requirements listed above.
- To answer the theory questions.

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

```
TEST PHASE 0: Polynomial - conversion constructor, conversion to std::string, and operator<<
p1 = 6.60
p2 = 0.00

TEST PHASE 1: Polynomial - constructors, conversion to std::string, and operator<<
p1 = 2.20 + 4.40 * X^1 - 2.00 * X^2 + 5.00 * X^3

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, conversion to std::string, set_base, and operator<<
0.00 + 1.00 * Log_2( 0.00 + 1.00 * X^1 )

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!!
```