Exercise session 04

Inheritance and polymorphism in C++.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

19 Oct 2023

Exercise 1: automatic differentiation (1/2)

Implement a C++ framework for computing derivatives of arbitrary functions using a polymorphic approach. The goal is to create a structure that can handle common arithmetic operations, such as addition, subtraction, multiplication, division, and exponentiation, on values and their derivatives.

Test the program to compute the value and the derivative of the polynomial $f(x) = 2x^3 - 3x^2 + 4x - 5$ and of the function $g(x) = \frac{1}{x^2}$ at $x = 2$.

Exercise 1: automatic differentiation (2/2)

1. Define an abstract base class `ADExpression` with two pure virtual functions:
   - `double evaluate()`: This function returns the value of the variable.
   - `double derivative()`: This function returns the derivative of the variable.
2. Implement a concrete class `Scalar` that inherits from `ADExpression`. This class represents a scalar variable and its derivative.
3. Implement the following operation classes that also inherit from `ADExpression`:
   - `Sum`: Represents the addition of two `ADExpression` objects.
   - `Difference`: Represents the subtraction of two `ADExpression` objects.
   - `Product`: Represents the multiplication of two `ADExpression` objects.
   - `Division`: Represents the division of two `ADExpression` objects.
   - `Power`: Represents raising an `ADExpression` object to a constant exponent.
4. In the `main` function, demonstrate the usage of these classes.

Exercise 2: inheritance and polymorphism for data analysis

In the context of data analysis, create a C++ program that models different types of data sources and transformation objects.

1. Define an abstract class `DataSource` with attributes and methods that represent common properties of data sources. Create derived classes such as `FileDataSource` and `ConsoleDataSource`.
2. Define an abstract class `DataTransformer` with a virtual method for data transformation. Create derived classes such as `LinearScaler`, `LogTransformer`, and `StandardScaler` that implement specific data transformation methods.
3. Test all functionalities by prompting the user with proper questions.

Exercise 2 (1/5)

1. Define an abstract class `DataSource` with a string attribute `name`, a vector `data`, a method `display_info()` and a pure virtual method `read_data()`.
2. Implement a constructor and a virtual destructor in the `DataSource` class with.
3. Create derived classes `FileDataSource` and `ConsoleDataSource` that inherit from `DataSource`.
4. Implement constructors for all classes to model different data source types. For example, `FileDataSource` should initialize a `filename` and an input file, `ConsoleDataSource` should have a default constructor.
5. Implement destructors for the derived classes. For example, the `FileDataSource` constructor should open the file, and its destructor should close it.

Exercise 2 (2/5)

1. Implement the `read_data()` methods. `FileDataSource::read_data()` should import values from a file (see `data.txt` as an example), whereas `ConsoleDataSource::read_data()` should read a list of values from the standard input.
2. Create objects of these classes and demonstrate inheritance. For example, create a `FileDataSource` object and call `display_info()` and `read_data()` methods to read and display data from a file.
3. Ensure that resources associated with data sources are properly managed during construction and destruction.

Exercise 2 (3/5)

1. Define a base class `DataTransformer`, bound to `DataSource` by polymorphic composition. `DataTransformer` should have a pure virtual method `transform()` that transforms the `data` vector in the corresponding `DataSource`.
2. Create derived classes `LinearScaler`, `LogTransformer`, and `StandardScaler` that inherit from `DataTransformer`.
3. Override the `transform()` method in each derived class to provide specific data transformation. For example, `LinearScaler` scales the data by multiplying them by a given scaling factor, `LogTransformer` applies a logarithmic transformation and sets negative entries to `0`, and `StandardScaler` performs standardization to the $[0, 1]$ interval.
4. Create objects of these classes and demonstrate their use to transform the previously defined `DataSource` object.

Exercise 2 (4/5)

1. Use the `DataSource` and the `DataTransformer` hierarchy polymorphically. In particular, the program should prompt the user to import data either from a file or from console, and to select the transformation method.
2. Display the original and the transformed data.

Exercise 2 (5/5)

Possibilities for extensions

1. In case a `FileDataSource` is selected, prompt the user to specify the filename from the console.
2. In case a `LinearScaler` is selected, prompt the user to specify the scaling factor from the console.
3. Add a new class to the `DataSource` hierarchy to import a given field from an input `CSV` file (see `data.csv` as an example).
4. Write a class or method to overwrite the original data from the text or `CSV` file with the transformed ones.