Dirk Praetorius
Michele Ruggeri

Summer semester 2019
June 14, 2019

Scientific programming in mathematics

Exercise sheet 12

Inheritance

Exercise 12.1. Explain the differences between `public`, `private`, and `protected` inheritance on the basis of a suitable example.

Exercise 12.2. Implement the class `Person`, which contains the data members `name` and `address` (of type `string`). Derive from `Person` the class `Student` that contains the additional data fields `student_number` (`int`) and `study` (`string`). Derive from `Person` also the class `Employee` that contains the additional data fields `salary` (`double`) and `job` (`string`). Write mutator methods, constructors, and destructors for all classes. Implement the method `print` for the base class `Person`. The method should print to the screen name and address of a person. Redefine this function for the derived classes `Student` and `Employee` so that also the additional data fields of these classes are printed. Test your implementation appropriately!

Exercise 12.3. Write a class `Account`, which provides the basic functionalities to manage a bank account. The data members of the class are the account number (`int`), the current balance in EUR (`double`), and the fee in EUR (`double`, the amount of money that a customer has to pay, e.g., yearly, to have an account). Implement constructors, destructor, and mutator methods to work with the class, as well as the following methods:

- `void deposit(double sum)`, to deposit a positive amount of money to the bank account;
- `void withdraw(double sum)`, to withdraw a positive amount of money from the bank account;
- `void chargeFee()`, to charge the yearly fee for the bank account;
- `void print()`, to print to the screen the information (account number, balance).

Using `Account` as base class, implement two derived classes: `SavingsAccount` and `CurrentAccount`. In addition to the attributes of an `Account` object, a `SavingsAccount` object contains a variable for an interest rate (`double`) and a method `void addInterest()`, which computes and adds the matured interest to the account. In addition to the attributes of an `Account` object, a `CurrentAccount` object contains an overdraft limit variable (`double`). Redefine the methods of the `Account` class if necessary in both derived classes. Test your implementation appropriately!

Exercise 12.4. Write a class `Bank` to store a dynamic array of `Account` objects (use the class and the derived classes of Exercise 12.3). The accounts in the array could be instances of the `Account` class, the `SavingsAccount` class, or the `CurrentAccount` class. Besides the usual functionalities to work with the class, implement methods for opening and closing accounts and the method `void updateAccounts()`, which iterates through all accounts and updates them in the following ways: All accounts are charged with the fee; Savings accounts get interest added (via the method you already wrote); Current accounts get a letter sent if they are in overdraft (print a message to the screen). To realize this, implement suitable methods `void update()` for the class and the derived classes of Exercise 12.3 which are then called by `updateAccounts`. Test your implementation appropriately!

Exercise 12.5. In mathematics, quaternions are an extension of complex numbers (which in turn extend real numbers). A quaternion is an expression of the form $a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$, where a, b, c, d are real numbers and $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are symbols that can be interpreted as unit-vectors pointing along the three spatial axes. Write a class `Quaternion` which contains the four coefficients a, b, c, d as data members (`double`) and the following functionalities: constructor, destructor, copy constructor, assignment operator, and mutator methods to work with the class. Moreover implement the method `double norm() const`, which computes and returns the norm of a quaternion. This is defined as

$$\|q\| = a^2 + b^2 + c^2 + d^2 \quad \text{for } q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}.$$

Finally, overload the operator `<<` in order to be able to print to the screen a quaternion $q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ stored in an object `q` of type `Quaternion` by typing `cout << q`. Test your implementation appropriately.

Exercise 12.6. We consider quaternions stored as objects of the class `Quaternion` from Exercise 12.5. Overload the sign operator `-`, which, applied to a quaternion $q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$, returns the quaternion $-q = -a - b\mathbf{i} - c\mathbf{j} - d\mathbf{k}$. Overload the tilde operator `~`, which, applied to a quaternion $q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$, returns its conjugate $\overline{q} = a - b\mathbf{i} - c\mathbf{j} - d\mathbf{k}$. Test your implementation appropriately!

Exercise 12.7. Overload the operators $+$ and $*$ in order to be able to calculate the sum and the product of two quaternions $q_1 = a_1 + b_1\mathbf{i} + c_1\mathbf{j} + d_1\mathbf{k}$ and $q_2 = a_2 + b_2\mathbf{i} + c_2\mathbf{j} + d_2\mathbf{k}$, stored as objects of type `Quaternion` from Exercise 12.5. The sum of two quaternions is performed componentwise, i.e.,

$$q_1 + q_2 = (a_1 + a_2) + (b_1 + b_2)\mathbf{i} + (c_1 + c_2)\mathbf{j} + (d_1 + d_2)\mathbf{k}.$$

An explicit formula for the product of two quaternions, usually referred to as Hamilton product, can derived from the product rules for the basis elements, i.e.,

$$\mathbf{i} \cdot 1 = 1 \cdot \mathbf{i} = \mathbf{i}, \quad \mathbf{j} \cdot 1 = 1 \cdot \mathbf{j} = \mathbf{j}, \quad \mathbf{k} \cdot 1 = 1 \cdot \mathbf{k} = \mathbf{k}, \quad \mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{i} \cdot \mathbf{j} \cdot \mathbf{k} = -1,$$

and the distributive law. Is the Hamilton product of quaternions commutative? Test your implementation appropriately!

Exercise 12.8. Complex numbers can be identified as quaternions of the form $q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ with $c = d = 0$. Real numbers can be identified as quaternions of the form $q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ with $b = c = d = 0$. Using the class `Quaternion` from Exercise 12.5 as base class, implement the derived class `Complex` to store complex numbers. Redefine all methods/operators from Exercise 12.5–12.7 (if needed) so that all functionalities work also for objects of type `Complex`. Moreover, implement the type castings `Quaternion` $\rightarrow$ `Complex` and `Complex` $\rightarrow$ `double`. Test your implementation appropriately!