Object-oriented and softw. design - C++

Lab worksheet #5 - Report

Hacini Malik, Boyer Timothé, Lainé Martin

November 2025

Contributions:

- Exercise 1: Martin
- Exercise 2: Martin & Malik
- Exercise 3: Timothé

N.B. — Doxygen documentation for the lab can be generated with the following commands:

- `sudo apt update` (if necessary)
- `sudo apt install -y doxygen graphviz` (if necessary)
- `make docs`
- `firefox docs/html/index.html`

Polymorphism

1. Why is polymorphism useful?

- **Uniform Handling:** Allows storing different account types (`Current`, `Blocked`, etc.) in a single array of base pointers (`Bank_account*`).
- **Runtime Binding:** Ensures the specific method (e.g., `Current_account::print`) is called automatically for a generic pointer.
- **Extensibility:** New account types can be added without modifying the code that manages the collection.

2. Required Modifications

In `account.hpp` (Base Class)

1. **Pure Virtual Methods (`= 0`):**
   - Declare `debit` and `print` as pure virtual (e.g., `virtual double debit(...) = 0;`).
   - **Reason:** Makes `Bank_account` abstract; forces derived classes to define their own logic.

Polymorphism

2. **Virtual Destructor:**
   - `virtual ~Bank_account() = default;`
   - **Reason:** Ensures derived class destructors are called when deleting a base pointer, preventing memory leaks.

In `account.cpp` & Derived Classes

1. **Global `operator<<`:**
   - Define it to take `const Bank_account&` and call `acc.print(os)`.
   - **Reason:** Acts as a wrapper to trigger the virtual `print` method.

2. **Derived Classes:**
   - Use `override` for `debit`/`print` and provide concrete implementations.

Client update

1. Dynamic Data Structure

To provide flexibility, we replaced fixed account attributes with a **dynamic array of pointers**: `Bank_account** accounts`.

- **Polymorphism:** This structure allows a single `Client` to hold a variable number of accounts of different types (`Current`, `Blocked`, etc.) mixed in the same array.

2. Memory Management (Rule of Three)

Since we use raw pointers with dynamic allocation (`new`), manual memory management is critical to prevent leaks and crashes.

- **Destructor:** We implemented a custom destructor that first deletes each individual account object (`delete accounts[i]`) and then deletes the array itself (`delete[] accounts`).
- **Deep Copy:** A copy constructor was implemented to perform a **deep copy**. It allocates a new array and duplicates each account object.

Client update

3. Interactivity

The `createAccount` method introduces runtime decision-making. It uses user input (`cin`) to instantiate the specific subclass (`new Blocked_account`, etc.) requested by the user and stores it in the polymorphic array.

Polynomials - 1

We define a new abstract class `Function` representing a mathematical function. This class provides pure virtual methods for evaluating the function at a given point, computing its derivative, and displaying the informations about the function.

```
class Function {
    protected:
        std::string name;
    public:
        Function(const std::string& i_name) : name(i_name) {}
        virtual
        double eval(double x0) = 0;
        virtual
        Function* derivative() = 0;
        virtual
        void display() = 0;
};
```

Polynomials - 2

We implement the derivative class `Poly0`, `Poly1`, and `Poly2` representing polynomials of degree 0, 1, and 2 respectively. Each class implements the methods defined in the `Function` class. They have more specific attributes for each coefficient of the polynomial. For the derivative method each class returns a new instance of the appropriate polynomial class representing the derivative.

In the `main` function, we create three instances of these polynomial classes and demonstrate polymorphism by storing them in a vector of `Function*`. We then display the informations about each polynomial and evaluate the derivative of each polynomial at the point $x = 1$.