REPORT - HPC TP2

Romain Jochum

Introduction

This report summarizes the work I implemented for Questions Q1 through Q4. The goal was to design an `Animal` base class (Q1), extend it with derived classes (Q2), enable polymorphism (Q3), and collect animals in an STL `vector` with helper utilities to inspect and total weights (Q4). I also compiled and ran small demos to verify behavior.

Q1 — Base `Animal` (file: Animal.h, used by q1.cpp)

What I implemented

- Extracted a single base class `Animal` into Animal.h to allow reuse across exercises.
- Kept core data members protected so derived classes can access them while preserving encapsulation from external code.
- Exposed public accessors (getters/setters) for external use.

Key design choices

- `protected` members:
  - `std::string name;`
  - `double weight;`
  - `std::string movement;`

    These are shared implementation details that derived classes can sensibly access/modify.
- `public` interface:
  - Constructor/destructor, `getName()`, `getWeight()`, `setName()`, `setWeight()`, `setMovement()` — these form the stable API for users.
- Virtual functions to allow polymorphism:
  - `virtual void print() const;`
  - `virtual void moveDescription() const;`
  - `virtual void type() const;`
- Concrete helper:
  - `void info_displacement()` implemented inline in header; it calls the virtual functions so it behaves polymorphically.

Representative extract from Animal.h

```
class Animal {
protected:
    std::string name;
    double weight;
    std::string movement;

public:
    Animal(const std::string &name_, double weight_, const std::string &movement_);
    virtual ~Animal() = default;
    std::string getName() const { return name; }
    double getWeight() const { return weight; }
    void setMovement(const std::string &m) { movement = m; }

    virtual void print() const {
        std::cout << "Name: " << name << ", Weight: " << weight << "kg, Movement: " <<
movement;
    }
    virtual void moveDescription() const { std::cout << movement; }
    virtual void type() const { std::cout << "animal"; }

    void info_displacement();
};
```

Why these choices?

- `virtual` allows derived overrides and polymorphic calls through `Animal*`.
- `protected` is a pragmatic choice for a teaching example: derived classes can use name/weight/movement without needing accessors everywhere, while external code still uses public API.

Q2 — Derived classes (each class now in its own header/source pair under `src/`)

What I implemented

- Derived classes: `Mammal`, `Bird`, `Fish`. Each adds members appropriate to its category and overrides the virtual functions.
- Bonus derived classes: `Penguin`, `Ostrich` (both derive from `Bird`), and `Dolphin` (derives from `Mammal`).

Representative extracts (from q2.cpp / questions.cpp)
Mammal:

```
class Mammal : public Animal {
protected:
    int numLegs;
    double noseLength; // cm

public:
    Mammal(const std::string &name_, double weight_, const std::string &movement_,
           int legs_, double noseLen_)
        : Animal(name_, weight_, movement_), numLegs(legs_), noseLength(noseLen_) {}

    void print() const override {
        Animal::print();
        std::cout << ", Type: mammal, Legs: " << numLegs << ", Nose length: " <<
noseLength << "cm";
    }
    void type() const override { std::cout << "mammal"; }
};
```

Bird (two constructors: full and convenience):

```
class Bird : public Animal {
protected:
    double wingSpan; // m
    bool canFly;

public:
    // Full form (movement + canFly)
    Bird(const std::string &name_, double weight_, const std::string &movement_,
         double wingSpan_, bool canFly_)
        : Animal(name_, weight_, movement_), wingSpan(wingSpan_), canFly(canFly_) {}

    // Convenience: assume flying and a default movement string
    Bird(const std::string &name_, double weight_, double wingSpan_)
        : Animal(name_, weight_, "flies around"), wingSpan(wingSpan_), canFly(true) {}

    void print() const override {
        Animal::print();
        std::cout << ", Type: bird, Wingspan: " << wingSpan << "m, Can fly: "
                  << (canFly ? "yes" : "no");
    }
    void type() const override { std::cout << "bird"; }
    void moveDescription() const override {
        if (canFly) std::cout << "flies with wings";
        else std::cout << movement;
    }
};
```

Fish:

```
class Fish : public Animal {
protected:
    bool saltwater;
    double length; // cm
public:
    Fish(const std::string &name_, double weight_, const std::string &movement_,
         bool saltwater_, double length_)
        : Animal(name_, weight_, movement_), saltwater(saltwater_), length(length_) {}

    void print() const override {
        Animal::print();
        std::cout << ", Type: fish, Habitat: " << (saltwater ? "saltwater" :
"freshwater")
                  << ", Length: " << length << "cm";
    }
    void type() const override { std::cout << "fish"; }
    void moveDescription() const override { std::cout << "swims"; }
};
```

Design choices explained

- Each derived class has `protected` members that represent additional attributes (e.g., `wingSpan`, `numLegs`) so they are available to further derived classes (Penguin/Ostrich).
- All overrides use the `override` specifier — this helps catch mistakes if the base signature changes.
- The constructor patterns:
    - Full constructors (take movement string and flags) for precise control.

- Convenience constructor for quicker instantiation in examples (e.g., `Bird("Coco", 1.5, 3.0)`).

Q3 — Enable polymorphism (demonstration in questions.cpp)

What I did

- Ensured `Animal` virtual functions exist and derived classes use `override`.
- Demonstrated calling overrides via base-pointer:

```
Animal* pA = new Bird("Coco", 1.5, 3.0);
pA->print();   // calls Bird::print()
delete pA;
```

Why this works

- `print()` (and `moveDescription()` / `type()`) are `virtual` in `Animal`. The derived classes implement `override`. Thus calls through `Animal*` dispatch to the derived class at runtime.

Q4 — Collection of animals and helpers (files: Zoo.hpp, Zoo.cpp, demo in questions.cpp)

What I added

- Zoo.hpp (declarations)

```
#ifndef ZOO_HPP
#define ZOO_HPP
#include "Animal.h"
#include <vector>
double total_weight(const std::vector<Animal*>& zoo);
void info_displacement(const std::vector<Animal*>& zoo);
void inventory(const std::vector<Animal*>& zoo);
#endif
```

- Zoo.cpp (implementations)

```
double total_weight(const std::vector<Animal*>& zoo) {
    double sum = 0.0;
    for (Animal* a : zoo) {
        if (a) sum += a->getWeight();
    }
    return sum;
}

void info_displacement(const std::vector<Animal*>& zoo) {
    for (Animal* a : zoo) if (a) a->info_displacement();
}

void inventory(const std::vector<Animal*>& zoo) {
    for (Animal* a : zoo) if (a) { a->print(); std::cout << std::endl; }
}
```

Demo usage (in questions.cpp)

```
std::vector<Animal*> Zoo;
Zoo.push_back(new Bird("Coco", 1.5, 3.0));
Zoo.push_back(new Fish("Dory", 0.3, "", true, 10.0));
Zoo.push_back(new Mammal("Wide_Ethelbert", 70.0, "runs", 2, 10.0));
Zoo.push_back(new Penguin("Eiscue", 25.0, 0.8));
Zoo.push_back(new Ostrich("Bipbip", 90.0, 2.1));

info_displacement(Zoo);
std::cout << "Total weight of the animals in the zoo is " << total_weight(Zoo) <<
std::endl;

// free memory
for (size_t i = 0; i < Zoo.size(); ++i) delete Zoo[i];
```

Notes on memory management and safety

- For the exercise I used `new` and `delete` as requested. This is fine for small demos, but in production code prefer smart pointers:
  - `std::vector<std::unique_ptr<Animal>>` — prevents leaks even on exceptions and removes manual `delete`.
- `Zoo.cpp` iterates with non-const `Animal*` so it can call non-const methods such as `info_displacement()` (which was implemented non-const in the header). If you prefer, `info_displacement()` could be marked `const` on `Animal` and then `Zoo.cpp` could use `const` iteration.

Build & test steps I ran

Commands used locally to validate:

```
# compile questions + zoo (now using per-class sources)
g++ -std=c++17 -Wall -Wextra -I src src/Questions.cpp src/Zoo.cpp src/Mammal.cpp
src/Bird.cpp src/Fish.cpp src/Penguin.cpp src/Ostrich.cpp src/Dolphin.cpp -o
build/Questions

# run
./build/Questions
```

Observed output included per-animal displacement lines and the total weight printed. Example final line:

```
Total weight of the animals in the zoo is 186.8
```

Conclusion

I implemented a small, clear OO hierarchy:

- `Animal` base class with `virtual` behavior
- `Mammal`, `Bird`, `Fish` derived classes, and `Penguin`, `Ostrich`, `Dolphin` bonus classes
  Then I demonstrated polymorphism (Q3) and used an STL `vector<Animal*>` to collect animals and compute a total weight (Q4).
  Finally I kept a simple and explicit memory model for the exercise (new/delete), and recommended moving to smart pointers for robustness as seen in the course.