CompSys202/MechEng270
Object-oriented design and programming
Assignment #1 (15% of final grade)
Due: 11:00am, 17 August 2016

Learning outcomes

The purpose of this assignment is to target the following learning outcomes:

- Gain confidence modeling a problem.
- Gain confidence programming in C++.
- Gain confidence using Linux.
- Apply OOP concepts such as inheritance, polymorphism and encapsulation.
- Apply good coding practices such as naming conventions and code reuse.

1 The Zoo Management System

In this assignment you will simulate the management of a zoo. Your program will allow a zoo to manage the animals by providing the following functionality:

- Keep track of the zoo's bank balance:
  - The balance increases as more visitors come to the zoo.
  - The balance decreases as the zoo purchases more food.
- Manage feeding of the animals:
  - Manage food stocks.
  - Feed the animals.

The command-line interface for the application has been implemented for you, but you must implement the functionality of the system itself.

2 Files provided

You have been given headers with class and method declarations for the classes you must implement in parts of this assignment. You are encouraged to add helper methods or data members to these classes where necessary to structure and simplify code, but you should not change any existing method declarations (except where specified).

Makefile This Makefile will allow you to build and run both the Zoo Management System command-line interface, and the automated tests for the Zoo Management System. The main commands relevant to this file are:

`make` Builds the zms and zms_test executables.

`make test` Builds the zms_test executable and runs it.

`make run` Builds the zms executable and runs it.

`make clean` Removes all executable files from the working directory.

main.cpp Defines the command-line interface of the Zoo Management System (ZMS), all of which has been implemented for you. Do not modify this file.

test.cpp This file contains some tests to make sure you have implemented some parts of the assignment correctly. It does not contain tests for the whole assignment, so don't assume that because all of the tests pass you will pass the Assignment! You are encouraged to add more tests for parts of the assignment not already covered, but the tests will not be marked at all. To run all of the tests execute the command `make test` from a command-line terminal. You are also welcome to share your **tests** on Canvas if you like.

Zoo/ZooManagementSystem.hpp Defines the class interface for the ZMS, used by both the command-line interface and unit tests. You may add to this file, but do not change the existing declarations.

Zoo/Animal.hpp Defines the interface for the `Animal` pure virtual base class, which all animal classes must inherit from. You may add to this file, but do not change the existing declarations. The animals which derive from `Animal` have been declared for you in Monkey.hpp, Lion.hpp, Elephant.hpp, Snake.hpp and Otter.hpp located inside the Zoo directory.

Zoo/Food.hpp Defines the interface for the `Food` class, which represents a type of food that may be fed to animals. You may add to this file, but do not change the existing declarations.

All other files in the Zoo directory are stubbed implementation files for the classes you must implement. You are free to modify them as you see fit. Particular sections of these files you are expected to modify to complete this assignment are marked with comments such as:

```
// TODO: Implement
```

See the Tasks section for more information about completing these implementation files.

3 Tasks

3.1 Implement the Food class *[Task 1]*

You will need to complete the implementation of the `Food` class by modifying the files Zoo/Food.hpp and Zoo/-Food.cpp. You may add more code to the header, but don't change the existing method declarations. The `Food` class holds the properties of a food item that the zoo has in stock. Every food item has a name, a cost, and a count of how many are left (the quantity). More food can be purchased, and food can be consumed by animals.

Testing the Food Class

The tests for the Food class are already enabled in test.cpp. To run the tests, run the command `make test` from a command-line terminal. The `test` target in the `Makefile` builds the executable `zms_test` from the code in test.cpp and runs it, executing all enabled tests. If you need to run a specific test you can run `./zms_test 3` to run only the third test (replace 3 with the number of the test that you want to run).

The test executable will report any test failures and print out information about why the test failed. For example:

```
FAILURE: test.cpp:64
    Expression 'banana.getQuantity() == 0' evaluated to false
FAIL: Test 1 failed.
```

In this example the test executable reports that the test is failing because the return value of `banana.getQuantity()` is not equal to `0`. If we inspect the test code (test.cpp line 64, according to the error message) we can see the context for this message:

```
/*
Test Food quantity is initially set to 0.
*/
TestResult test_FoodQuantity() {
    Food banana("banana", 4.50, 250);
    ASSERT(banana.getQuantity() == 0); // <--- failing here

    return TR_PASS;
}
```

So the test is creating a new food type called "banana" with a cost of $4.50 and an energy content of 250kJ, and then checking that the quantity of the food available is initially set to `0`. Thinking back to how the `Food` class was coded, you might discover that the quantity was set to the wrong initial value in the `Food` class constructor (or that it was never set at all!):

```
Food::Food(const string& new_name, double new_cost, unsigned int new_energy) {
    this->quantity = 2;
}
```

Once the error is fixed by setting the initial value of the quantity property to `0`, the test will pass:

```
INFO: Executing test 1
PASS: Test 1 passed.
```

Making sure that your program passes all of the tests you have been given will not guarantee that you will get 100% of the marks for this assignment, but it will mean that you get at least some marks for the parts that work according to those tests. Writing your own tests or extending the ones you have been given is strongly recommended, as there are many requirements in this assignment that the existing tests do not cover.

3.1.1 `Food(string& new_name, double new_cost, unsigned int new_energy)`

Implement the constructor of the `Food` class. The three parameters passed into this method refer to the name, cost and energy content of the food. You need to set the instance variables name and cost respectively from these parameters.

3.1.2 `getName()`, `getCost()`, `getQuantity()`, `getEnergy()`

These methods should return the respective instance variable; the food's name, cost per unit, energy content (all set in the constructor) and the quantity remaining.

3.1.3 `unsigned int consume(unsigned int count)`

This method will decrease the quantity of the food by the amount specified by `count`. If there is less food available than the value of `count`, consume all of it and set the remaining quantity to `0`. The `unsigned int` value returned is the number of units of this food that were consumed; note that this may be less than the count that was given via the `count` parameter.

3.1.4 `void purchase(unsigned int count)`

This method will increase the quantity of the food remaining by the specified `count`.

3.2 Implement the Animals *[Task 2]*

The `ZooManagementSystem` won't be of any use without some animals to manage. You will complete the `Monkey`, `Lion`, `Elephant`, `Snake` and `Otter` classes defined in the respective header files within the Zoo folder (e.g. Zoo/Monkey.hpp). The implementation files for each class have also already been created for you (e.g. Zoo/Monkey.cpp) with blank implementations for all of the methods. Each animal will have different hunger levels and food preferences, according to the following table:

Animal	Type	Daily Food Intake (kJ)	Required Foods
Monkey	`AT_MONKEY`	800	banana, watermelon, mealworms
Lion	`AT_LION`	2200	steak, mouse, fish
Elephant	`AT_ELEPHANT`	8400	banana, watermelon, hay
Snake	`AT_SNAKE`	250	mouse, egg
Otter	`AT_OTTER`	750	fish, mouse

Table 1: Animal behavior for different Animal subtypes

Each `Animal` subclass must implement the full `Animal` interface with the correct behavior for that animal. This means you must implement the following methods **for each animal** (Monkey, Lion, Elephant, Snake, Otter):

`virtual AnimalType type() const` Return the correct `AnimalType` enumeration value (defined within the `Animal` class) for each animal from this method, e.g. `AT_LION` for the `Lion` class.

`virtual unsigned int hungerLevel() const` Return the current hunger level of the animal, measured in the number of kiloJoules (kJ) the animal animal needs to eat in the current day. This will be equal to the animal's daily food intake requirement, less the energy value of the food the animal has already eaten.

`virtual void resetToHungry()` Resets the animal's `hungerLevel` to the original level (the animal's *Daily Food Intake* requirement).

`virtual bool likesFood(const Food& food)` Indicates whether an animal likes the given food. Returns true if the animal will eat the food, or false if it will not. The foods each animal likes are listed in the *Required Foods* column of Table 1 on page 3. You can check the identify of a food using the `getName` method of the `Food` class.

`virtual int feed(Food& food)` Feed a given amount of a specified food to the animal. The `Animal` will eat the specified food until it is full. The `Animal` will stop eating if there is not enough of the food available, or if it becomes full before eating all of the food (its `hungerLevel` reaches zero). If an animal has less hunger remaining than the energy that would be provided by eating a whole unit of food, it still consumes a whole unit of food to reduce the `hungerLevel` to `0`.

You may implement these methods in the individual animal subclasses or in the `Animal` base class. If you implement methods in the `Animal` base class you will have to delete the code for the declarations and implementations of those methods in the derived classes so they aren't overridden. Consider how you can use this to create good Object Oriented design when implementing the animals.

Testing the Additional `Animal` Subclasses

The tests for these additional `Animal` subclasses can be enabled by uncommenting the following line at the top of test.cpp:

```
// #define ENABLE_T2_TESTS
```

Once the Task 2 tests are enabled, run `make test` to execute all of the currently enabled tests. For more details on testing this assignment, refer to section 3.1. Remember that not all aspects of these classes are tested in the tests you have been given, so **don't assume the task is complete once all of the tests pass**.

3.3 Implementing the Zoo Management System *[Task 3]*

The `ZooManagementSystem` class implements the core functionality of the system. It ties together the main data groups (animals and food) and controls interaction between them. It also manages the zoo's balance, as a side effect of buying food and admitting visitors. Note: don't implement the `feedAllAnimals` method of the `ZooManagementSystem` yet, as this will be done in Implement Automatic Feeding *[Task 4]*. Modify the Zoo/ZooManagementSystem.hpp and Zoo/ZooManagementSystem.cpp files to implement the `ZooManagementSystem` class (remember you may not change the class name or any of the given method declarations in Zoo/ZooManagementSystem.hpp).

Testing the `ZooManagementSystem`

Tests for the `ZooManagementSystem` class have been included in the test.cpp file provided to you. To enable them, uncomment the following line at the top of test.cpp:

```
// #define ENABLE_T3_TESTS
```

This will enable the tests for the `ZooManagementSystem` class. Save the test.cpp file and run `make test` to execute all of the currently enabled tests. For more details on testing this assignment, refer to section 3.1.

3.3.1 `static string author()`

This static method should return your username (UPI) as a string, e.g. "ngia003".

3.3.2 `ZooManagementSystem(double initialBalance)` and `double getBalance()`

The `ZooManagementSystem` constructor is used to set the zoo's initial bank balance at startup. After construction, the `getBalance` method is used to check the current balance. Add a member variable of type `double` to the `ZooManagementSystem` class and implement both the constructor for the `ZooManagementSystem` and the `getBalance` method.

3.3.3 `double admitVisitors(unsigned int adults, unsigned int children)`

The `admitVisitors` method should handle the admission of visitors to the zoo. Its two parameters are the number of adults and children in the group being admitted. Using this information it:

- Calculates the total admission cost; $14 per adult, $5 per child.
- Adds the total admission cost to the zoo's balance.
- Returns the total admission cost so it can be printed to the screen.

The zoo also recently introduced a family pass system, where groups of exactly 2 adults and 2 children can be bundled together into a family pass for a total cost of $30. A group may have multiple family passes applied to discount their entry if there are enough adults and children. For example:

- A group of three children and two adults would qualify for one family pass, so the total cost of their visit to the zoo would be $35.
- A group of four children and four adults would qualify for two family passes, so the total cost for this group would be $60.
- A group of three adults and one child wouldn't qualify for a family pass, so their total cost would be $47.

3.3.4 `vector<Food> getFood()` and `void addFood(const Food& new_food)`

These functions are the basis for getting and adding food types. `getFood` returns a vector of all the food items the zoo currently stocks (even if the zoo has run out of units of a food, it technically still stocks it) in the order the food items were added using `addFood` (least recently added to most recently added). It is therefore advisable that you use a vector to store all of the `Food` items within the `ZooManagementSystem` class, so you can directly return a copy of the vector from this method. `addFood` should add new `Food` items to the `ZooManagementSystem`, i.e. they should be included in the vector returned by `getFood`.

3.3.5 `vector<Animal*> getAnimals()` and `bool addAnimal(Animal::AnimalType new_animal_type)`

Much like the `getFood` method, the `getAnimals` method returns a vector of animals in the order they were added using `addAnimal` (least recent to most recent). However, this method returns a vector of *pointers* instead of returning a vector of concrete object instances. Polymorphic objects must always be accessed through a reference or a pointer. Pointers and references are a constant size, no matter what type of object they point to. So although the size of the underlying storage for individual objects may differ, pointers to all objects are the same size. Vectors and arrays can only store objects of the a uniform size (and type), so objects of different sizes or types can't be stored in the same vector. We *can* however store polymorphic base class pointers which point to instances of different classes, as these are all the same type and size.

> In reality, a C++ compiler might let you get away with storing instances of a polymorphic class directly in a vector (depending on the types involved), but it causes the objects to be copied from their original full-size storage area to a region of memory that will only fit base-class-sized object. For all intents and purposes the copied object will be an instance of the base class, not the original subclass type. This may sometimes work by pure luck, and other times it will cause data loss and strange behavior in your program. This effect is called object slicing.

The `addAnimal` method is used to create new animals, which are then added to the animals vector within the `ZooManagementSystem` class. Use the `new` keyword to create these new animals on the heap. This method should create different animals depending on the value of the `new_animal_type` parameter. The return value of the `addAnimal` method is a boolean value indicating whether the requested animal was created and added successfully. If the `new_animal_type` parameter is set to `AT_INVALID` or another value which does not correspond to an animal type, `addAnimal` should return `false`. If an animal is successfully created and added, it should return `true`.

3.3.6 `void resetAllAnimals()`

Implement the `resetAllAnimals` method which should reset the hunger level of all animals by calling each animal's `resetToHungry` method.

3.3.7 `~ZooManagementSystem()`

Implement the `ZooManagementSystem` destructor so it uses `delete` to properly clean up the animal objects created in the `addAnimal` method. The `ZooManagementSystem` is responsible for deleting all heap-allocated `Animal`s from memory.

3.3.8 `bool purchaseFood(double& cost, unsigned int index, unsigned int quantity)`

The `purchaseFood` method is used to increase the quantity available of a food in the zoo's stores by purchasing more stock. The `index` parameter indicates the index of the food to be purchased within the food vector returned by `getFood`. The `quantity` parameter is the number of units of food that should be purchased. The cost parameter is actually a return value. This is an example of common pattern for returning multiple values in C and C++ called an **output parameter**: a reference or pointer to a value outside the function is used to return more information than can be communicated via the return value alone. To return a value via an output parameter, simply set that parameter's value to the value you want to return. Output parameters must be pointers or references.

Implement this method by using the index to find the food item in your food vector and then use the `purchase` method of the food object to purchase more stock. If the purchase is successful this method should return true, and it should set the cost output parameter to the total cost of the purchase. If the food index given is invalid (e.g. outside the range of the food vector) this method should return `false` and set the `cost` output parameter to `0.0`.

The zoo doesn't have any credit facility on its main account, so the bank balance should never become **negative** because of a food purchase. If a purchase would cause the balance to become negative, the purchase should be rejected by returning `false` (the balance should be unchanged). However, in this case the cost of the purchase should still be correctly calculated, and should be returned via the `cost` output parameter.

Make sure there is enough money in the zoo's balance for the purchase **before** subtracting it from the balance; note the cases where a purchase can fail and make sure your `purchaseFood` method returns the correct values in these cases.

3.3.9 `FeedResult feedAnimal(unsigned int& quantity_eaten, unsigned int animal_index, unsigned int food_index)`

Implement this method to allow users to feed animals. Animals are identified by their index in the vector returned by `getAnimals`, so you can use the `animal_index` parameter to find the animal to be fed. Likewise the `food_index` refers to an item from the food vector returned by `getFood`. Use the `Animal::feed` method to feed the food to the animal. The return value of `feedAnimal` depends on the outcome of the feeding process:

- If there was a large enough quantity of the selected food available and the animal was fully fed, `FR_SUCCESS` should be returned (even if the food was exhausted after the animal was fully fed).
- If the `animal_index` parameter is invalid (outside the range of the elements in the animals vector), `FR_INVALID_ANIMAL` should be returned.
- If the food_index parameter is invalid (outside the range of elements in the food vector), `FR_INVALID_FOOD` should be returned.
- If both the animal and food indexes are invalid, return `FR_INVALID_ANIMAL`.
- If the animal doesn't want to eat the food return `FR_INCOMPATIBLE`.
- If the food was exhausted **before** the animal could be fully fed, return `FR_EXHAUSTED`. If the food was exhausted but the animal was still fully fed return `FR_SUCCESS`.

The `quantity_eaten` parameter of `feedAnimal` is an output parameter; it should be set to the number of units of food the animal consumed in the feeding process (i.e. `0` if no food was consumed). For more information on output parameters, see the documentation for the `purchaseFood` method in section 3.3.8.

3.3.10 Our first application: running the Zoo Management System!

Once you have implemented the `ZooManagementSystem` class you can test it yourself by running the ZMS command-line interface. The `zms` application can be run from the command line after executing `make`. To build and run the application in a single step, a `run` target has been included in the `Makefile`; so you can build and run the command-line interface for the Zoo Management System with the `make run` command. Note that if you haven't yet finished tasks 1 and 2 the Zoo Management System won't function properly (but it should compile and run).

After asking you for the initial bank balance, the Zoo Management System will present a main menu with several options, which you can select by typing the number corresponding to the menu entry and then pressing the Enter key. Each menu option will exercise part of the `ZooManagementSystem` class. You should explore all of the options and exercise each option to test that your code works properly.

```
Please select from one of the following options:
    0: QUIT
    1: Display the zoo's available balance
    2: Admit visitors to the zoo
    3: Display the available food stock
    4: Purchase food
    5: Display the animals currently held by the zoo
    6: Add an animal to the zoo
    7: Feed the animals
    8: Reset the hunger levels for all animals

 >>>
```

3.4 Implement Automatic Feeding *[Task 4]*

Do not attempt this task until you have completed Tasks 1-3.

In the existing implementation of the `ZooManagementSystem` the user must select an appropriate food to feed each `Animal`. If there is not enough of the food in stock to feed the `Animal`, the user must keep feeding the `Animal` with other foods. It is also possible for the user to pick foods which the animal doesn't like.

To solve these issues with the application, you must implement a simple automatic feeding method, where the `ZooManagementSystem` can use its models of animal behavior to try and figure out which foods each animal will eat and automatically select the correct food to feed each animal. In order to optimize how the zoo's food stores are used, your automatic feeding method should:

- Always feed the hungriest animal first.
- Feed animals the food that the zoo has the most of, out of all the foods the animal likes.
- Where two or more animals are equally hungry, determine the feed order by their index in the vector returned by `getAnimals`. Feed the animal with the lowest index first.
- Where two or more foods that an animal likes have the same quantity available, choose which one to use by their index in the vector returned by `getFood`. Use the food with the lowest index first.

If the zoo has no food left that an animal likes, the automatic feeding system should simply skip that animal.

This method will be declared in the `ZooManagementSystem` class, and will be called `feedAllAnimals`. This method has already been declared in ZooManagementSystem.hpp and a method stub has been created for you in ZooManagementSystem.hpp. Make sure that you **do not permanently reorder the animal or food vectors** in the `ZooManagementSystem` class, as the ordering specified in 3.3.4 and 3.3.5 must be maintained.

It is *strongly* recommended that you create one or more automated tests in test.cpp to make sure this method works correctly, as it will not be testable through the current `zms` command line interface or the unit tests you have already been given.

Important: how your code will be marked

- Your code will be marked using a semi-automated setup. If you fail to follow the setup given, your code **will not be marked**. All submitted files must compile without requiring any editing. Use the provided tests and `Makefile` to ensure your code compiles and runs without errors. Any tests that run for longer than 10 seconds will be terminated and will be recorded as failed.

- Although you may add more to them (e.g. member variables, `#include` statements, or helper functions), you must not modify the interface of classes defined in the following files (e.g. do not delete the existing functions declared):
  - `Zoo/ZooManagementSystem.hpp`
  - `Zoo/Animal.hpp`
  - `Zoo/Food.hpp`
- Do not move any existing code files to a new directory, and make sure all of your new code files are created inside the `Zoo` directory.
- You may modify `main.cpp` and `test.cpp` as you please (for your own testing purposes); these files will not be marked at all. Be aware that your code must still work with the original `main.cpp` and `test.cpp`.
- Your code will also be inspected for good programming practices, particularly using good object-oriented principles. Think about naming conventions for variables and functions you declare. Make sure you comment your code where necessary to help the marker understand **why** you wrote a piece of code a specific way, or **what** the code is supposed to do. Use consistent indentation and brace placement.

Submission

You will submit on the Assignment Drop Box, by visiting `https://adb.auckland.ac.nz`. Make sure you can get your code compiled and running with both main.cpp and test.cpp (`make run` and `make test`). Submit the following, in a single ZIP archive file:

- A **signed and dated declaration** stating that you worked on the assignment independently, and that it is your own work. Include your name, ID number, the date, the course and assignment number. All code will be checked against other submissions. Submissions detected as being similar to others will ensure that the students involved are forwarded to the Misconduct Committee.
- The entire contents of the src_for_students folder you were given at the start of the assignment, including the new code you have written for this assignment. Ensure you **execute `make clean` before zipping the folder** so your submission doesn't include any executable files (your code will be re-built for marking).

Academic honesty

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help. Under no circumstances should you take or pay for an electronic copy of someone else's work.
- All submitted code will be checked using software similarity tools. Submissions with suspicious similarity will result in an Investigative Meeting and will be forwarded to the Disciplinary Committee.
- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
  - Always do individual assignments by yourself.
  - Never show or give another person your code.
  - Never put your code in a public place (e.g. Reddit, Github, forums, your website).
  - Never leave your computer unattended. You are responsible for the security of your account.
  - Ensure you always remove your USB flash drive from the computer before you log off.

Late submissions

Late submissions incur the following penalties:

- 15% penalty for zero to 24 hours late
- 30% penalty for 25 to 48 hours late
- 100% penalty for over 48 hours late (dropbox automatically closes)

You must double check that you have uploaded the correct code for marking! There will be no exceptions if you accidentally submitted the wrong files, regardless of whether you can prove you did not modify them since the deadline. No exceptions.