Raphael Braun
Ayesha Feroz
Kyowon Jeong
Matteo Pilz
Lukas Ruppert
Arslan Siraj
Faezeh S. Zakeri

Eberhard Karls
Universität
Tübingen

Programming in C++

Sheet 4

Submission date: 07.10.2022, 12:00

The Amoeba Game
In this simple game, the player is an amoeba that fights and eats other microorganisms. The DNA of these microorganisms is used for levelling up the amoeba which makes it become stronger and stronger. However, enemies also get stronger every round. The goal for the player is to survive as long as possible. In this worksheet we will practice several concepts like function overriding, polymorphism and passing lambda functions to create the amoeba as well as its enemiesfood.

4.1 Polymorphism, abstract classes and lambda functions (100 Points)

C++

We will start with the base class **Food**. Both player and enemy will derive from it later.

a) Implement the **is_alive** of the **Food** class. It should return true if `health > 0`, otherwise false.

b) Implement the **constructor** of the **Food** class at the position indicated in the code. The constructor should initialize the class member variables with the passed values. We will use this constructor later to create enemies of different strength.

Now we will implement different classes of enemies. Let's assume that our game design already tells us that we need to add a lot of customization later. Thus, we decide to create one C++ class for each class of enemies.

a) Complete the **DeadCell** class, which is derived from **Food** class: First, complete the **constructor** which takes (health, power, defence) as arguments. `health_gain=10.0, dna_gain=100.0` should be initialized by calling the constructor of the base class.

b) Create and implement the **Bacterium** class analogous to DeadCell (implement constructor same as in DeadCell with function arguments (health, power, defence), and (health_again=10.0, dna_gain=200) passed to the base class).

c) Create and implement the **Virus** class with same as constructor define above in (DeadCell, Bacterium) classes, the default values will be (health_gain=400.0, dna_gain=400.0).

Hint: The Bacterium and Virus class, implements same functions as DeadCell class (available in student_template), and make sure cmake picks these classes by adding it to the CMakeList.
Now let's implement the Amoeba class (yes, that is us: the player). Your task is to implement the missing parts of the **Amoeba** class (which is also derived from the **Food** class).

a) Add a protected member variable of type double with name `dna_level_th` with 100.0 as default value to the **Amoeba** class. This will be used to determine if our player level's up. Complete the default and custom constructor. We don't want to become food of some enemy early on so, by default, we give us a good advantage `health=1000.0, power=50.0, defence=50.0, dna_level_th=100.0`. Note that custom the Amoeba constructor has different parameters and we will use it later to initialize our amoeba with non-default values.

b) Override the **print** and **print_header** member functions in the **Amoeba** class. Instead of the gain variables it should print out the `dna_level_th` and the `dna_level`.

c) Implement the **eat** member function of the **Amoeba** class. The method eat adds `health` and `dna` to its respective member variables. If `dna_level` is larger or equal to the threshold `dna_level_th`, then (1) `power` is increased to `power += random_value() * power`, (2) the `dna_level_th` is doubled, and (3) `dna_level` reset to zero.

Now read the **engine.h** function and fill the missing pieces:

a) Create one instance of the **Amoeba** class in the **engine** function with the defaults we defined above.

b) In each round, we will fill the `all_enemies` vector in the **engine** function with exactly one DeadCell, one Bacterium, and one Virus. Our enemies are born with certain skills and aptitudes. For dead cells (which should be an easy enemy, right) we consider `health=10.0, power=0.0, defence=0.0` the default for creating DeadCell objects in **engine.h**. Bacteria are our first serious enemy, and we consider `health=200.0, power=10.0, defence=50.0` a default for them. A Virus is a significantly stronger enemy with defaults `health=500.0, power=25.0, defence=25.0`. We will use the random_value function to create some variation between enemies and to also make them stronger (more health, more power, more defence) each round. When creating the object, instead of passing the defaults for health, power and defence as listed above, we pass `random_value(difficulty)*default health, random_value(difficulty)*default power, random_value(difficulty)*default defence` upon construction for each type of enemy.

c) For educational purposes, we will take a look at a common (and often required) technique that allows us to perform deep polymorphic copies (=make a deep copy given just a `Food*`). (If you know Java you might have seen the related Java clone() methods before.) Implement the **clone** member function of the **Food, DeadCell, Bacterium, Virus, and Amoeba** class, such that it returns a copy of the object as `Food*`. Hint: Don't copy the pointer (shallow copy) but dynamically create the object using the (synthesized) copy constructor and return it as `Food*`. Fill the `enemies` vector in the **engine** function with 1 to 3 randomly sampled (e.g. use `std::rand`) (deep) copies from the `all_enemies` vector. Duplicates are allowed and should show up with same stats. Don't forget to delete the allocated memory at the end and clear the vectors after each round.

d) Implement the **combat** function in the **engine.h**. For educational purposes, we did not add a setter for the health function. Instead, we provide a member `attacked` that takes a lambda function that determines if the attack was successful and updates the protected variables. During the first attack, the enemy is attacked by the player. The lambda function that you need to implement (1) computes `random_value()*player→get_power() - random_value()*enemy→get_power()`. (2) checks if this value is positive, and if so subtracts the value from the enemy health. The second attack, where the player is attacked, works analogously just the other way around.

The game ends if the **Amoeba** is dead and the number of rounds (=the maximum difficulty) is printed.