UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program Design: Introduction

Practical 5 Specifications: Inheritance

Release date: 25-09-2023 at 06:00
Due date: 29-09-2023 at 23:59
Total Marks: 337

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually, no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- The following imports are allowed ("<iostream>", "<sstream>", "<string>" ,"<stdlib.h>", "<cmath>", "Brute.h", "Mage.h", "Ninja.h", "Soldier.h","Sword.h","ShadowBone.h"). You do not need all of these, however any imports apart from these will result in a mark of 0.
- Be ready to upload your assignment well before the deadline, as **no extension will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or classes).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at http://www.ais.up.ac.za/plagiarism/index.htm.

- Unless otherwise stated, the usage of c++11 or additional libraries outside of those indicated in the assignment, will not be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use c++98**
- We will be overriding your .h files, therefore do not add anything extra to them that is not in the spec
- In your student files you will only find a skeleton main
- If you would like to use *using namespace std*, please ensure it appears in your cpp that you upload

2 Plagiarism

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding.** Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

The goal of this practical is to gain experience with working with inheritance.

4 Introduction

Implement the UML diagram and functions as described on the following pages.

5 Attack Rules

Before starting with the *attack* methods, there are a few rules that will be mentioned once, but all attack methods need to follow:

The method must do nothing and return false if any of the following is true:

- The pointer sent in is NULL.
- The opponent is already dead.
- This object does not have a sword set. With the exception of Mage, as Mage can attack with just ShadowBone, but not with no swords.

If there is a sword(s) set, then use the *damage*() (taking in the player strength) function of the sword to get the amount of damage and the *takeDamage*() function to apply damage to the opponent. If you manage to kill the opponent, return true.

A rule of thumb in the attack methods:

- **Never modify the strength member variable**
- When you need to modify how much strength is used, such as in the Brute class, apply these steps. For the Ninja and Mage classes, only steps 2 and 3 are needed.
  1. Calculate the strength needed for that attack.
  2. Call the *damage*() function of the sword using the strength calculated, this will return how much damage the sword has done.
  3. Call *takeDamage*() of your opponent using the amount of damage that was just calculated.

6 Class Diagram

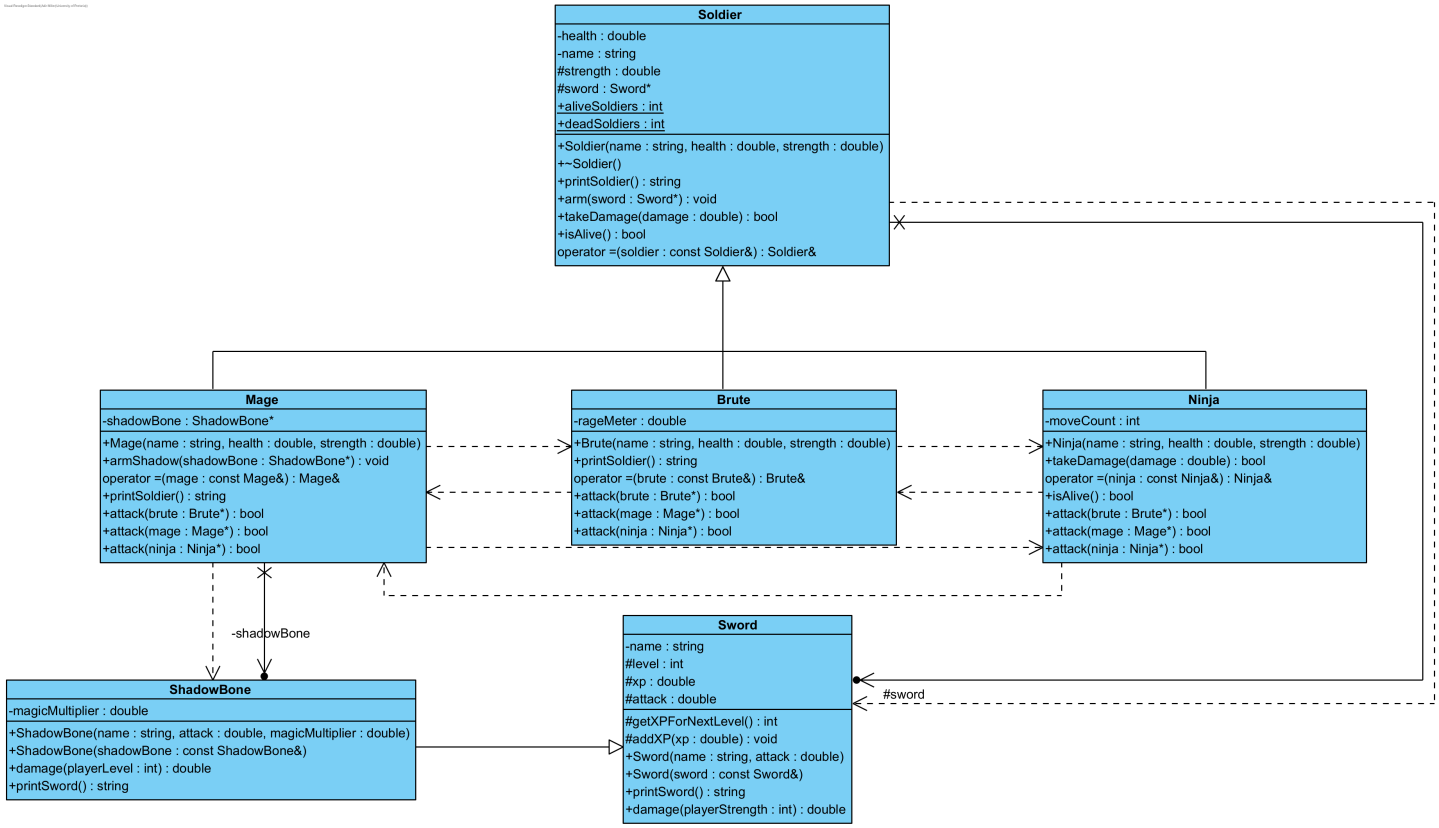

Figure 1: Class diagrams

7 Classess

7.1 Sword

- Members
  - name: string
    - A private string variable containing the name of the sword.
  - level: int
    - A protected int member variable indicating the level of the sword.

  - xp: double
    - A protected double member variable indicating the XP of the sword.
  - attack: double
    - A protected double member variable indicating the attack power of the sword.

- Functions
  - Sword(name: string, attack: double)
    - This constructor should set all member variables, including setting *xp* to 0 and *level* to 1.
  - getXPForNextLevel(): int
    - This function returns the needed xp for the sword to level up.
    - The following equation should be used: $ln(2 \times (level + 1))^3$.
    - Remember, the natural log *ln* in cmath is *log*() and to raise to the power its *pow*().
  - addXP(damage: double): void
    - This function adds XP to the sword, it should add to the *xp* member variable.
    - It should add 10% of the damage that was sent in, so if 100 damage was sent in, 10 XP should be added.
    - If *xp* is now larger then *getXPForNextLevel*(), *level* should be incremented and *xp* should be reset to 0.
    - Even if there is enough XP to jump 2 levels, only increment once, and set *xp* to 0.
  - damage(playerStrength: int): double
    - This function will return how much damage the sword did, this will be calculated as follows:
      $$attack + (attack \times (playerStrength \times 0.1 + level \times 0.2))$$
    - This function should also call *addXP*() and send in the amount calculated above.
  - printSword(): string
    - This function should print out a sword in the following format, *sstream* may be used to convert double and int to string but, **NO** formatting should be applied regarding the decimal places.

```
Sword: CoolSword\n
Attack: 78\n
Level: 80\n
XP: 25.5\n
```

  - Sword(sword: const Sword&)
    - Simple copy constructor for sword.

7.2 ShadowBone: public Sword

- Members
  - magicMultiplier: double
    - A private double member variable indicating the *magicMultiplier*

- Functions
  - ShadowBone(name: std::string, attack: double, magicMultiplier: double)

    - Constructor should set all variables including *magicMultiplier*.
  - ShadowBone(shadowBone: const ShadowBone&)
    - Simple copy constructor for ShadowBone
  - damage(playerLevel: int): double
    - This function does everything its parent does, but with one change to the equation:

$$attack + (attack \times ((playerStrength \times 0.1 + level \times 0.2) + magicMultiplier))$$

  - printSword(): string
    - This function does everything its parent does, but at the end it should add the magicMultiplier in the following format:

```
Magic Multiplier: 5\n
```

7.3 Soldier

- Members
  - health: double
    - Private double member variable indicating the soldier's health.
  - name: string
    - Private string member variable indicating the soldier's name.
  - strength: double
    - Protected double member variable indicating the soldier's strength.
  - sword: Sword*
    - Protected Sword pointer to a **dynamic** sword variable.
  - aliveSoldiers: static int
    - A static int variable indicating the number of soldiers alive.
  - deadSoldiers: static int
    - A static int variable indicating the number of dead soldiers.
- Functions
  - Soldier(name: string, health: double, strength: double)
    - The constructor for the Soldier class.
    - This constructor should increment the *aliveSoldiers* variable.
    - The constructor should initialize all variables.
  - ~Soldier()
    - The destructor for the soldier class.
    - If a sword is set, it should be deallocated.
    - Whether the soldier possesses health or not, this function should **NOT** decrement **aliveSoldiers** nor increment **deadSoldiers**.
  - printSoldier(): string
    - This function returns a string of the soldier, it does NOT print anything to the console.
    - You may use sstream to convert the doubles to strings.

    - Do not change any decimal places when returning.
    - The following is the format it should be printed out, this is the format if the soldier without a sword. If they do have a sword then the result *printSword*() should be appended.

```
Name: Jeff\n
Health: 198.6\n
Strength: 5.56\n
```

  - arm(sword: Sword*): void
    - This function sets the *sword* member variable.
    - If *sword* is already set, it should be deallocated before the new one is set.
    - If the passed in parameter is the same as the member variable, the function should do nothing.
    - Nothing should happen if NULL is passed in.
  - takeDamage(damage: double): bool
    - This function returns a boolean, based on whether or not the damage done is enough to kill the soldier.
    - Nothing should happen and false should be returned if *damage* is negative.
    - If the damage is enough to kill the soldier, then *aliveSoldiers* should be decremented and *deadSoldiers* should be incremented.
    - If after the damage has been applied, *health* is smaller than 0, *health* should be set to 0, as a soldier cannot have negative health.
    - If this function is called on a dead soldier, it should simply return false.
  - isAlive(): bool const
    - Note this is a const function, **not** a function returning a const bool.
    - Returns **true** if *health* is larger than 0 and **false** otherwise.
  - operator=(const Soldier& soldier): Soldier&
    - A standard assignment operator, remember to use deep copies.

7.4 Mage : public Soldier

- Members
  - ShadowBone* shadowBone
    - A private pointer to a **dynamic** ShadowBone Object.
- Functions
  - Mage(name: string, health: double, strength: double)
    - The constructor for the Mage class.
  - armShadow(shadowBone: ShadowBone*): void
    - This function should set the *shadowBone* member variable.
    - This function should follow the same rules given for *arm*().
  - operator=(mage: const Mage&): Mage&
    - Assignment operator for the Mage class.
  - printSoldier(): string

        * This function calls its parents *printSoldier*(), and appends *printSword*() of *shadowBone* if it is set before returning the string.
    - attack(brute: Brute*): bool
        * This follows listed Attack rules (see section 5)
        * First try and use *shadowBone* if it is set, if that manages to kill the brute then return true and do not try to use the *sword*.
        * If *shadowBone* does not manage to kill the brute, try and use the *sword*.
        * If *shadowBone* is not set, only use *sword*.
    - attack(ninja: Ninja*): bool
        * This function follows all the same rules as *attack(brute Brute*)*.
    - attack(mage: Mage*): bool
        * A mage cannot use a ShadowBone on another mage, therefore it may only use a *sword*.

7.5 Brute : public Soldier

Attack rules:
Brutes have a rage meter. Every time they **DO NOT** manage to kill an enemy, they get angrier which increases their rage meter. If they get a kill, then their rage meter is reset to 0. If the attack is called and their rage meter is equal to 10 or higher, the following equation is used to calculate their strength used for the attack:

$$strength \times 2 + strength \times (rageMeter - 10) \times 0.1$$

.

- Members
    - rageMeter: double
        * A private double member variable indicating the rage meter.
- Functions
    - Brute(name: std::string, health: double, strength: double)
        * Constructor for the Brute class.
    - printSoldier(): string
        * This function calls its parents *printSoldier*(), and appends the rage meter in the following format:

```
Rage meter: 3.5\n
```

    - operator=(brute: const Brute&): Brute&
        * The assignment operator for the Brute class.
    - attack(mage: Mage*): bool
        * Standard attack rules with the rules given above (just under the Brute: public Soldier heading) and in the *Attack Rules* (see section 5).
    - attack(ninja: Ninja*): bool
        * Standard attack rules with the rules given above (just under the Brute: public Soldier heading) and in the *Attack Rules* (see section 5).

- attack(brute: Brute*): bool
  - Standard attack rules with the rules given above (just under the Brute: public Soldier heading) and in the *Attack Rules* (see section 5).
  - One change comes in the form that a brute can only attack another brute with half its strength. When it comes to the rage meter, FIRST calculate the strength using the equation given above, and only then half it before you send it into the *damage*() function.

7.6 Ninja : private Soldier

- Members
  - moveCount: int
    - A private int member variable indicating the moveCount.
- Functions
  - Ninja(name: string, health: double, strength: double)
    - The constructor for the Ninja class.
  - takeDamage(damage: double): bool
    - This redefines the *takeDamage*() function of its parent. A ninja will not take damage every third call, this means each time this function is called *moveCount* should be incremented. If *moveCount* reaches 2, no damage should be taken and *moveCount* should be reset to 0.
  - operator=(ninja: const Ninja&): Ninja&
    - Assignment operator for the Ninja class.
  - isAlive(): bool
    - Redefined isAlive(), this is needed since its private inheritance.
  - attack(brute Brute*): bool
    - Simple attack using the attack rules listed in *Attack rules*.
  - attack(ninja: Ninja*): bool
    - Simple attack using the attack rules listed in *Attack rules*.
  - attack(mage Mage*): bool
    - Simple attack using the attack rules listed in *Attack rules*.

8 Memory Management

Memory management is a core part of COS110 and C++, so this practical memory management is extremely important due to the scope. Therefore each task on FitchFork will allocate approximately 20% of the marks to memory management. The following command is used:

```
valgrind --leak-check=full ./main
```

Please ensure, at all times, that your code *correctly* de-allocates *all* the memory that was allocated.

9 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the practical marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the main.cpp file) that will be used to test the Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing, the gcov [1] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j Sword ShadowBone Soldier Mage Ninja Brute
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

We will scale this ration based on class size.
The mark you will receive for the testing coverage task is determined using table 1:

Coverage ratio range	% of testing mark	Final mark
0%-5%	0%	0
5%-20%	20%	6
20%-40%	40%	12
40%-60%	60%	18
60%-80%	80%	24
80%-100%	100%	30

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the Instructor Provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

10 Upload checklist

The following c++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- main.cpp
- Ninja.cpp
- Brute.cpp
- Mage.cpp
- Soldier.cpp
- Sword.cpp
- ShadowBone.cpp

[1] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

You will notice you do not need to upload a makefile or any h files, you may if you wish, however, FitchFork will overwrite them before running your submission.
The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

11 Submission

You need to submit your source files, only the cpp files (**including the main.cpp**), on the FitchFork website (https://ff.cs.up.ac.za/). All methods must be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. There is no need to include any other files or .h files in your submission. **Ensure your files are in the root directory of the zip file**. Your code should be able to be compiled with the C++98 standard
For this practical you will have 10 upload opportunities. Upload your archive to the Practical 5 slot on the FitchFork website.