UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science
COS110 - Program Design: Introduction
Practical 1

1 Introduction

Deadline: 3rd September, 19:30

1.1 Objectives and Outcomes

The objective of this practical is to test your understanding of the programming concepts covered in the theory classes. In particular, this practical will test your understanding of classes and pointers in terms of typical usage.

1.2 Submission

All submissions are to be made to the **ff.cs.up.ac.za** page under the COS 110 page, and for the correct practical slot. Submit your code to Fitchfork before the closing time. Students are **strongly advised** to submit well before the deadline as **no late submissions will be accepted**.

1.3 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 textual material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to **http://www.ais.up.ac.za/plagiarism/index.htm** (from the main page of the University of Pretoria site, follow the *Library* quick link, and then click the *Plagiarism* link). If you have questions regarding this, please ask one of the lecturers, to avoid any misunderstanding.

1.4 Implementation Guidelines

Follow the specifications of the practical precisely. For each practical, you will be required to create your own makefile so pay attention to the names of the files you will be asked to create. If the practical requires you to submit additional files of your own, follow the file structure and format exactly. Incorrect submissions will use up your uploads and no extensions will be given. In terms of C++, unless otherwise stated, the usage

of C++11 or additional libraries outside of those indicated in the practical, will not be allowed. Some of the appropriate files that you submit will be overwritten during marking to ensure compliance to these requirements. If the specification makes use of text files, for providing input information, be sure to include blank text files with the specified names.

1.5 Mark Distribution

Activity	**Mark**
Factory Class	40
Drone Class	10
Total	**50**

2 Practical

2.1 Classes -Constructors and Destructors

Constructors and destructors are important parts of fully realising the functionality provided by classes. In the case of constructors, they let a wide range of information be provided at class instantiation so that the use of additional setter functions can be avoided. It also provides flexibility to a class because multiple constructors, each taking a different set of arguments,can be defined and implemented under the same name, a practice called overloading. Destructors on the other hand, are meant to facilitate the process of object clean up. Specifically, destructors exist to clear up the memory used by the class when it is instantiated and used as an instance. The more complex the memory requirements of a class, the more complex the destructor needs to be. By default, constructors and destructors are provided but this practical aims to introduce user defined constructors and destructors alongside more complex class usage. From now on, destructors are noted specifically by the usage of "$\sim$". Constructors are also unique in that they do not have a return type specified.

Additionally, you will be not be provided with mains. You must create your own main to test that your code works and include it in the submission which will be overwritten during marking.

2.2 Task 1

In this task you are going to implement a part of a program for an auto-battler game. There are two classes **factory** and **drone**. The first class represents a player controlled factory and the second class describes the drones that the player creates in the factory. Each of these classes is defined in a simple UML diagram below:

2.2.1 Factory Class

```
factory
-list: drone **
```

```
-fID:string
-oID: string
-level: int
-droneLimit: int
-currNumDrones:int
----------------------------
+factory(fID:string, oID:string, droneLimit:int)
+factory(fID:string, oID:string,drones: drone**, droneLimit:int,
currNumDrones:int)
+~factory()
+levelUp():void
+addDrone(d:drone*):int
+removeDrone(name:string):int
+getFID():string
+getOID():string
+getCurrSize():int
+getLimit():int
+printDrones(s:string):void
```

The variables of the class are as follows:

- list: A dynamic array of drone objects.
- fID: The ID of the factory.
- oID: The ID associated with the owner of the factory.
- level: The factory's current rank. This is a number with every value being associated with an $*$. For example a rank 2 factory would be $**$ and so on. By default, all factories start at rank 1.
- droneLimit: The number of drones the factory is allowed to create and store in the list variable.
- currNumDrones: The current number of drones in the factory's list. This can change over time. It will initially start at 0.

The methods defined for the class are as follows:

- factory(fID:string, oID:string, droneLimit:int): A constructor for the factory class. It sets the factory ID, owner ID and the droneLimit variable. The list variable should be initialised but not populated by any objects yet.
- factory(fID:string, oID:string, drones: drone**, droneLimit:int,currSize:int): A second constructor for the factory class. It sets the factory ID and owner ID . It also receives an array of drone objects and the necessary variables to instantiate the current object's list. The objects in the drones argument should be used to instantiate the list variable in the class, creating a copy.
- ~factory(): The class destructor. It deletes all of the memory created by the class. The list should be deleted from index 0 to the last index.

- addDrone(d:drone*): This function receives a drone object as an argument. This adds the argument into the list variable, adding it at the first available index. It returns the index at where the new creature was inserted. If the team is full, return -1. When added into the list, the drone's HP and energy should be multiplied by the current level of the factory.
- removeDrone(name:string): This removes a drone associated with the passed in argument. If there are multiple examples of the name, the first one found from index 0, should be removed. The memory associated with it should also be deleted. If the drone could not be found, return -1.
- getFID(): Returns the ID of the factory.
- getOID(): Returns the ID of the owner.
- getCurrSize(): Returns the current number of drones in the list variable.
- getLimit(): Returns the limit of the number of drones.
- levelUp(): This increases the level of the factory by 1.
- printDrones(s:string): Prints out the following information (with example values):

  ```
  Factory ID: 001
  OID: 001
  Number of Drones: 3
  Drone Limit: 10
  Rank: ***
  Energy Level: 1000
  Mega Tank Drone
  Heli-Drone
  Jet Drone
  ```

  After the rank is displayed, all of the drones should be displayed in the order indicated by the argument, s, passed in:

  - hp: Sort the list by hp ascending
  - m: Sort the list by energy descending

  The energy level output is the total energy value of every created drone in the list. If there are no drones, then no Energy Level value should be displayed.

2.2.2 Drone Class

```
drone
-name: string
-type: string
-hp: int
-energy:double
-----------------------------
+drone(name: string,type: string,hp: int,energy:double)
```

```
+~drone()
+getName():string
+getType():string
+getHP():int
+getEnergy():double
+print():void
```

The variables are as follows:

- name: The name of the drone.
- type: Their type of drone being built. The following types are available:
  - Air
  - Tank
- hp: The number of hit points the drone has.
- energy: The amount of energy the drone has. The energy is used to do attacks.

The methods have the following behaviour:

- drone(name: string,type: string,hp: int,energy:double): The constructor for the drone class. It instantiates the variables of the class with the given arguments.
- ~drone(): The destructor for the class. When called, it should print out a message with the following format (with a new line at the end):

  ```
  X deleted
  ```

  where X represents the name of the drone being deleted.
- getName():A getter for the name variable.
- getType():A getter for the type variable.
- getHP(): A getter for the HP variable.
- getEnergy(): A getter for the energy variable.
- print(): This function prints out the details of the drone in the following format (with example values):

  ```
  Name: Mega Tank Drone
  Type: Tank
  HP: 300
  Energy: 1000
  ```

  Each line of output should have a new line at the end.

You will be allowed to use the following libraries: **string, iostream**. You are allowed to use namespace standard in your header files. You will have a maximum of 10 uploads for this task. Your submission must contain **drone.cpp, drone.h, factory.cpp, factory.h, main.cpp** and a makefile.