UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

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

1 Introduction

Deadline: 18th of September, 18:00

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 inheritance in addition to other previously covered topics.

1.2 Submission

All submissions are to be made to the **assignments.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 elses work without consent, copying a friends 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**
Vehicle Class	10
Locomotive Class	10
DieselLocomotive and ElectricLocomotive Class	10
Total	**30**

2 Practical

2.1 Inheritance

Inheritance is a concept used in programming languages to greatly extend the usefulness of a class and also to better model some relationships that the class is meant to represent. In particular, C++ supports 6 types of inheritance: single, multi-level, multiple, hierarchical, hybrid and multipath inheritance. In the scope of this practical, we will be looking at hierarchical inheritance. Furthermore, C++ supposed 3 modes of inheritance: public, protected and private. You would do well to become familiar with how inheritance influences member access and so on. Methods which are virtual are presented in italics in UML and methods which are pure virtual are also set to 0, such as *foo()=0:void*.

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

Imagine that you are working for a logistics company operating out in the frontiers of civilisation. Your boss wants to be able to run some basic analytics on some of the routes that you currently support and wants you to implement some systems to help her. However the current limitations of the legacy system prevent you from making fundamental changes. Instead, you will need to use inheritance in order to achieve your goal.

2.2.1 Vehicle Class

The **vehicle** class is the parent class of a derived class: **locomotive**. Their inheritance will be public inheritance so reflect that appropriately in their .h files. The description of the **vehicle** class is given in the simple UML diagram below:

```
vehicle
-map: char**
-name: string
-size:int
-------------------------
+vehicle()
+setName(s:string):void
+getName():string
+getMap():char**
+getSize():int
+setMap(s: string):void
+getMapAt(x:int, y:int):char
```

+∼vehicle()
+operator−−():void
+determineRouteStatistics()=0:void

The class variables are as follows:

- map: A 2D array of chars, it will represent the map that each vehicle will have to travel on.
- name: The name of the vehicle. For example, "Frontier Express".
- size: The size of the map as a square matrix.

The class methods are as follows:

- vehicle: This is the constructor of the class. It is simply the default constructor with no additional features.
- getSize: This returns the size of the map as a square matrix.
- setName: This will set the name of the vehicle as received.
- getName: This will return the vehicle as set.
- getMap: This will return the entire map variable.
- setMap: This method receives the name of a text file that contains an ASCII map. The map will be a square, equal number of rows and columns. The first line of the map will have the number of rows. Every line after will contain a number of ASCII characters that you must read into the map. This must allocate memory before assigning the map.
- getMapAt: This receives two coordinates, an x and y, and returns what character is located at those coordinates. If the coordinates are out of bounds, return ':'.
- ∼vehicle: The destructor for the class. It has been made virtual.
- determineRouteStatistics: This function will be used to determine information from the map based on requirements specific to the vehicle in question. As it stands, it is made pure virtual.

- operator−−: The overload of this operator will deallocate the memory allocated for the map.

2.2.2 Locomotive Class

The description of the locomotive class is given by the simple UML diagram below:

```
locomotive
-supplyRange:int
--------------------------------
+locomotive()
+~locomotive()
+getSupplyRange():int
+setSupplyRange(s:int):void
+determineRouteStatistics():void
```

The class variables are as follows:

- supplyRange: This is the range, in terms of units (a unit being the distance between one map coordinate and another) that the locomotive can travel on one trip. The locomotive will require both coal and water in order to function and can only get it at supply locations on the map which are marked by a "*". A supply location will always be directly next to or adjacent to a railway, if one exists.

The class methods have the following behaviour:

- locomotive: The constructor of the class. It has no features beyond the default.
- ∼locomotive: This is the class destructor that will deallocate the memory assigned by the class. It will also print out, "locomotive removed", without the quotation marks and ended by a new line.
- getSupplyRange: Getters for the class variables.
- setSupplyRange: Setters for the class variables.
- determineRouteStatistics: This function needs to calculate the specific statistics for the locomotive based on the map it is provided. The following key shows all the specific elements that are pertinent to the locomotive:

  1. *: Supply Stations. These supply the locomotive with coal and water.
  2. O: Origin Points. This is where the trains will be expected to leave from.
  3. E: Exit Points. This is where the train is expected to go towards.
  4. #: Railroad. This is traversable tracks for the train. Locomotives can only travel on the map where there is track laid.

  The function will then determine a number of statistics and print them to the screen in a neatly formatted way:

1. Distance: Distance from the origin to exit in units, where one "#" is one unit so a track of "### " is a 3 unit long track.
2. Number of Supply Stations: Determine the number of supply stations that support the railway.
3. Journey Status: This will display: "Viable" or "Not Viable" depending on whether the locomotive is capable of making the journey. To determine this, calculate the distance from the origin to the exit as units. If the train can reach this distance with its current supply distance, then the trip is viable. If there is at least one supply station, then the trip will be viable. If not, it will not be viable.

Display the information as follows:

```
Name: Frontier Express
Supply Range: 12 units
Origin Coordinates: 1,2
Exit Coordinates: 8,7
Distance: 16
Number of Stations: 2
Status: Viable
```

Finally an example small map is provided below:

```
O#--
-#*-
-#--
-##E
```

As a note, the coordinates cannot be negative.

2.2.3 dieselLocomotive Class

The description of the dieselLocomotive class is given by the simple UML diagram below:

```
dieselLocomotive
-passengerLimit: int
---------------------------------
+dieselLocomotive()
+~dieselLocomotive()
+getPassengerLimit():int
+setPassengerLimit(s:int):void
+determineRouteStatistics():void
```

The class variables are as follows:

- passengerLimit: This is the limit, in terms of passengers, that a train can carry on a single trip.

The class methods have the following behaviour:

- dieselLocomotive: The constructor of the class. It has no features beyond the default.
- ~dieselLocomotive: This is the class destructor that will deallocate the memory assigned by the class. It will also print out, "diesel locomotive removed", without the quotation marks and ended by a new line.
- getPassengerLimit: Getter for the class variables.
- setPassengerLimit: Setters for the class variables.
- determineRouteStatistics: This function needs to calculate the specific statistics for the locomotive based on the map it is provided. The following key shows all the specific elements that are pertinent to the locomotive:

  1. M,N,P: Passenger Stations. The locomotive must pick up passengers from these stations. Each station will produce a certain quantity of passengers. These are
     (a) M: 50 passengers
     (b) N: 25 passengers
     (c) P: 10 passengers
  2. O: Origin Points. This is where the trains will be expected to leave from.
  3. E: Exit Points. This is where the train is expected to go towards.
  4. #: Railroad. This is traversable tracks for the train. Locomotives can only travel on the map where there is track laid.

  The function will then determine a number of statistics and print them to the screen in a neatly formatted way:

  1. Distance: Distance from the origin to exit in units, where one "#" is one unit so a track of "### " is a 3 unit long track. This does not include the origin and exit points. It will include the passenger stations.
  2. Journey Status: This will display: "Viable" or "Not Viable" depending on whether the locomotive is capable of making the journey. To determine if the journey will be viable, you need to determine if the train assigned to the map can carry enough passengers for what the route will typically entail at most. These stations are grouped into what they typically produce in volume so it is a reasonably safe bet to use their volume numbers when calculating. Basically, you must calculate how many passengers the locomotive will take on during its journey and if that number is greater than its carrying capacity, the journey is not viable. Otherwise it will be viable. You will need to display how many passengers the train is expected to pick up on its journey as well.

  Display the information as follows:

```
Name: Frontier Express
Origin Coordinates: 1,2
Exit Coordinates: 8,7
```

```
Distance: 16
Passengers Carried: 75
Status: Viable
```

Finally an example small map is provided below:

```
O#--
-M--
-P--
-##E
```

2.2.4 electricLocomotive Class

The description of the electricLocomotive class is given by the simple UML diagram below:

```
electricLocomotive
-perUnitCost: double
---------------------------------
+electricLocomotive()
+~electricLocomotive()
+getUnitCost():double
+setUnitCost(s:double):void
+determineRouteStatistics():void
```

The class has the following variables:

- perUnitCost: A double value showing the cost per each rail unit.

The class methods have the following behaviour:

- electricLocomotive : The constructor of the class. It has no features beyond the default.
- getUnitCost,setUnitCost: The getter and setter for the unit cost variable.
- ∼electricLocomotive : This is the class destructor that will deallocate the memory assigned by the class. It will also print out, "electric locomotive removed", without the quotation marks and ended by a new line.
- determineRouteStatistics: This function needs to calculate the specific statistics for the locomotive based on the map it is provided. The following key shows all the specific elements that are pertinent to the locomotive:

  1. O: Origin Points. This is where the trains will be expected to leave from.
  2. E: Exit Points. This is where the train is expected to go towards.
  3. #: Railroad. This is traversable tracks for the train. Locomotives can only travel on the map where there is track laid.

  The function will then determine a number of statistics and print them to the screen in a neatly formatted way:

1. Distance: Distance from the origin to exit in units, where one "#" is one unit so a track of "### " is a 3 unit long track. This does not include the origin and exit points.
2. Cost: This will display the cost of the rail construction. The cost of the rail is the number of tracks multiplied by perUnitCost. The cost should be converted into a whole integer, rounding down.

Display the information as follows:

```
Name: Frontier Express
Origin Coordinates: 1,2
Exit Coordinates: 8,7
Distance: 16
Cost: 160000
```

Finally an example small map is provided below:

```
O#--
-#--
-#--
-##E
```

You will be allowed to use the following libraries: **sstream,fstream, cstring, string, iostream**. Your submission must contain:

- vehicle.h
- vehicle.cpp
- locomotive.h
- locomotive.cpp
- dieselLocomotive.h
- dieselLocomotive.cpp
- electricLocomotive.h
- electricLocomotive.cpp
- map1.txt
- map2.txt
- map3.txt
- makefile
- main.cpp

You will have a maximum of 10 uploads for this task.