UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

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

1 Introduction

Deadline: 15th October, 20:40

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 **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

Task	**Mark**
Task 1	64
Total	**64**

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

Your task for this practical will be to make use of an inheritance hierarchy to simplify the development of a skeleton for a train designing program. The core of this system is a generalised component that can be added into a train. There are three types of components which have different roles and purposes but are similar enough that an inheritance hierarchy can be employed.

2.2.1 Hierarchy

The hierarchy of the classes is as follows. The base class is the **component** class. It has three children **passenger**,**cargo** and **locomotive**. All inheritance is public.

2.3 Component

The component class is given by the following UML diagram.

```
class component
-weight:double
-powerDrain:int
-double:cost
-char:type
---------------------------------------------
+component()
+component(type:char,weight:double,cost:double,powerDrain:int)
+getWeight():double
+getCost():double
+getPowerDrain():int
+getType():char
```

+∼*component()*
+*calculateEfficiency()=0:double*

The class variables are as follows:

- weight: The weight the component.
- powerDrain: The amount of power that the given component requires in order to operate.
- cost: The cost of the adding the component into a train.
- type: The type of component.

The class methods are as follows:

- component(): The default constructor for the component class. It does nothing.
- component(type:char,weight:double,cost:double,powerDrain:int):
- getWeight(): Getter for the weight variable.
- getCost(): Getter for the cost variable.
- getPowerDrain(): Getter for the powerDrain variable.
- getType(): Getter for the type variable.
- ∼*component()*: A virtual destructor for the class.
- *calculateEfficiency()*: A virtual function that will be used to calculate an efficiency score for the given component. It's made pure virtual.

2.4 Locomotive

The locomotive class is given by the following UML diagram.

```
class locomotive:public component {
-powerGen:int
-------------------------------------------------------------------
+locomotive(type:char,weight:double,cost:double,powerDrain:int,powerGen:int)
+getPowerGen():int
```

+~*locomotive()*
+*calculateEfficiency():double*

The class variables are as follows:

- powerGen: The amount of power a given locomotive generates.

The class methods are as follows:

- locomotive(type:char,weight:double,cost:double,powerDrain:int,powerGen:int)::the constructor for the locomotive class. When called it should instantiate all variables as provided in the arguments.
- getPowerGen(): A getter for the powerGen variable.
- ~*locomotive()*: A virtual destructor for the class.
- *calculateEfficiency()*: An implementation of the virtual function in the parent class. When called, it should return the efficiency score. This calculation is as follows:

$$(\frac{(powerGen)}{(powerDrain)})/cost \tag{1}$$

2.5 Cargo

The cargo class is given by the following UML diagram.

```
class cargo:public component {
-cargoCapacity:int
-------------------------------------------------------------------
+cargo(type:char,weight:double,cost:double,powerDrain:int,cargoCapacity:int)
+getCargoCapacity():int
```

+~*cargo()*
+*calculateEfficiency():double*

The class variables are as follows:

- cargoCapacity: The amount of cargo this component can carry.

The class methods are as follows:

- cargo(type:char,weight:double,cost:double,powerDrain:int,cargoCapacity:int): When called it should instantiate all variables as provided in the arguments.
- getCargoCapacity(): Getter for the cargoCapacity variable.
- *∼cargo()*: A virtual destructor.
- *calculateEfficiency()*:An implementation of the virtual function in the parent class. When called, it should return the efficiency score. This calculation is as follows:

$$(\frac{(weight + cargoCapcity)}{(powerDrain)})/cost \tag{2}$$

2.6 Passenger

The passenger class is given by the following UML diagram.

```
class passenger:public component {
-passengerCapacity:int
------------------------------------------------------------------
+passenger(type:char,weight:double,cost:double,powerDrain:int,passengerCapacity:int)
+getPassengerCapacity():int
```

+∼passenger()
+calculateEfficiency():double

The class variables are as follows:

- passengerCapacity: the number of passengers that can be carried.

The class methods are as follows:

- passenger(type:char,weight:double,cost:double,powerDrain:int,passengerCapacity:int): When called it should instantiate all variables as provided in the arguments.
- getPassengerCapacity(): Getter for the passengerCapacity variable.
- *∼passenger()*: Virtual destructor for the class.
- *calculateEfficiency()*:An implementation of the virtual function in the parent class. When called, it should return the efficiency score. This calculation is as follows:

$$(\frac{(weight * passengerCapacity)}{(powerDrain)})/cost \tag{3}$$

2.7 Train

The train class is given by the following UML diagram.

```
class train
-design:component **
-trainSize:int
-------------------------------------------------------------------
+train(size:int)
+~train();
+friend operator<<(output:ostream &,t:const train &):ostream &
+addComponent(comp:string):int
+removeComponent():int
```

The class variables are as follows:

- design: A dynamic array of components, this is going to represent the design of the train with each element referring to a component object.
- trainSize: A fixed size for how large the train is going to be.

The class methods are as follows:

- train(size:int): The constructor for the class. When called it assigns the variables and also initialises the design array with each element being set to NULL.
- ∼train(): The destructor for the class. When called it should deallocate all of the memory of the class and print out the following message:

  ```
  Number of Cargo Cars: A
  Number of Passenger Cars: B
  Number of Locomotives: C
  ```

  where A,B and C are the number of each type of component held in the design array.
- operator<<(output:ostream &,t:const train &): An overload of the outstream operator. When called it prints out the following message (with example values provided):

  ```
  DESIGN: pppl
  Efficiency Rating: 1.9601
  Cargo Cars: 0
  Passenger Cars: 3
  Locomotives: 1
  ```

  The message represents the design of the train in terms of the types of components, followed by the sum total efficiency rating of the entire train as well as the summary of the composition of the train.

- addComponent(comp:string): This function is used to add components into the design array of the train class. This function receives a string in with the format

  ```
  X,A,B,C,D
  ```

  where X represents the type of component and A–D are the input parameters required for instantiating an instance of the component required. The instance should be added at the first open space. It should return the index in the array where it was added. If the array is full then it should return -1.

  The types of components are as follows:

  - c: cargo
  - p: passenger
  - l: locomotive

  An example of this addition is

  ```
  c,90.1,775.1,14,1000
  ```

  This will add a cargo component with the weight, cost, powerDrain and cargo capacity specified afterwards.

- removeComponent(): This function removes the first component it finds in the design array. This component is deleted in memory. Remember to follow thorough memory management protocols when deleting components. Only one should be deleted per function call and it returns the index of the deleted component, with a -1 returned if none was deleted.

3 Submission Guidelines

The **train** class should make use of cstdlib,iostream and sstream as includes.

Your submission must contain:

- component.h
- component.cpp
- locomotive.h
- locomotive.cpp
- cargo.h
- cargo.cpp
- passenger.h
- passenger.cpp

- train.h
- train.cpp
- makefile
- main.cpp

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