UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

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

1 Introduction

Deadline: 1st of August, 07:00

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 classes and pointers in terms of typical usage.

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**
Order Class	20
Item Class	10
Total	**30**

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 be implementing part of a program used for online shopping. Specifically, you are going to implement the two classes, **item** and **order**, that are used to represent the order a customer can place with the service. The item class is specifically going to represent individual items with the order class being used to organise a collection of items for a single person. Each of these classes is defined in a simple UML diagram below:

2.2.1 Order Class

```
order
-items: item**
-customerName:string
-orderID: string
-orderSize: int
-currentSize: int
-----------------------------
+order()
+order(cName:string, cID:string, sizeOrder:int)
+order(cName:string, cID:string, itemsList: item**, sizeOrder:int, sizeList:int)
+~order()
+addItem(i:item*):int
+tallyCost():double
+getItems():item**
+getCustomerName():string
+getOrderID():string
+getCurrentSize():int
+getOrderSize():int
```

The variables of the class are as follows:

- customerName: The name assigned to the order based on the customer who placed it.
- orderID: A string of numeric characters that identify each order uniquely.
- items: a 1D array of pointers to objects of the class item. This is a list of items that is contained within the order class and represents the bulk of the functionality of the class.
- orderSize: This variable is the maximum number of items in the order.
- currentSize: This is the current number of items in the order. This can be less than the orderSize but cannot be over.

The methods defined for the class are as follows:

- order(): The default constructor. This must simply print the sentence, "order class constructed", without the quotation marks to the screen with a new line at the end.
- order(cName:string, cID:string, sizeOrder:int): This will receive three arguments, the customer name, order ID and the maximum size of the order respectively. Then it will create an instance of the order and instantiate the class with the passed in arguments. When created it will mean the item list is blank and so the currentSize should reflect that. The items variable is also instantiated but left as an array of the specified size.

- order(cName:string, cID:string, itemsList: item**, sizeOrder:int, sizeList:int): This will receive the customer name, order ID and a list of pre-created items with the maximum number of items specified as an argument. It must then create an instance of the order class with these arguments, copying the information contained within the list of items passed in as an argument. The variable sizeList refers to the size of the current itemsList.
- ~order: This is the destructor. It will deallocate the memory assigned to the items variable before the class is deleted. Be sure to make sure that the memory is allocated first before deallocating it. The final state of the variable should be NULL.
- addItem(i:item*): This method will simply copy the information contained with the argument passed in and add a new item to the array of items already maintained. It will return the index at which the new item was added. If the number of items is already equal to the size of the order, then the item is not added and the function returns -1. The new item added is always added in the next available position.
- tallyCost(): This function will calculate the cost of all the items in the order. This is the summation of every item's costPerItem multiplied by the quantity ordered. This is returned.
- getItems(): Returns the items variable
- getCustomerName(): Returns the customer name
- getOrderID(): Returns the order ID
- getCurrentSize(): Returns the current size of the order in terms of the number of items, just distinct items, contained within.
- getOrderSize(): Returns the maximum size of the order that is allowed.

2.2.2 Item Class

```
item
-itemName: string
-quantityOrdered: int
-costPerItem: double
----------------------------
+item()
+item(name:string, quantity: int,cost:double)
+~item()
+getItemName():string
+getQuantityOrdered():int
+getCostPerItem():double
+print():void
```

The variables are as follows:

- itemName: The individual name for the item

- quantityOrdered: The number of this specific item ordered. For example, the item might be a Chocolate Bar A, but 25 have been ordered of this one type.
- costPerItem: This is the specific cost for 1 quantity of this item. For example if the costPerItem is 5.00 and the quantity is 2, the total cost of this item is 10.00

The methods have the following behaviour:

- item(): The default constructor. This must simply print the sentence, "item class constructed", without the quotation marks to the screen with a new line at the end.
- item(name:string, quantity: int,cost:double): This constructor must instantiate an instance of the item class using the arguments provided.
- ∼item(): This is the default destructor. It simply prints "item deleted", without the quotation marks with a new line at the end.
- getItemName(): This returns the itemName.
- getQuantityOrdered(): This returns the quantity of the items ordered.
- getCostPerItem(): This returns their per unit cost.
- print(): This just prints out each of the values, each on a new line, stored in the item. For example:

```
Name: Potatoes
Quantity: 25
Cost Per Item: 1.25
```

You will be allowed to use the following libraries: **string, iostream**. You will have a maximum of 10 uploads for this task. Your submission must contain **item.cpp, item.h, order.cpp, order.h, main.cpp** and a makefile.