COS110 Assignment 1
Indiana Jones and his Friends

Due date: 19 August 2015
Total marks: 60

1 General instructions

• This assignment should be completed individually.

• 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 of certain functions or classes).

• Read the entire assignment thoroughly before you start coding.

• To ensure that you did not plagiarize, your code will be inspected with the help of dedicated software.

• 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.

2 Overview

For this assignment, you will create a world where adventurers look for ancient artefacts in the caves. If an artefact is old and precious, an adventurer may decide to sell it to a city museum. To determine the price of an artefact, a keen eye of a professional appraiser is necessary.
3 Your task

The goal of this assignment is to put you on first-name basis with C++ classes. You will create 6 classes that interact with one another via public interfaces and friend functions. Implement classes one by one, and test each class thoroughly before moving to the next one. Write a Makefile to make compilation and linking easier.

Classes that you will create (in the implementation order):

1. Artefact - artefact that an adventurer may find in a cave and sell to a museum
2. Inventory - a reusable class that stores a dynamic array of dynamically allocated artefacts
3. Cave - a cave where artefacts may be found
4. Museum - a museum that needs artefacts for the exhibition
5. Adventurer - an adventurer who seeks artefacts in the caves, and later sells them to museums
6. Appraiser - an appraiser who, given the age and material of an artefact, can estimate the price of the artefact

3.1 Artefact class (10 marks)

You have to write all code for this assignment yourself. For the Artefact class, create two files: Artefact.h (class definition) and Artefact.cpp (class implementation). The following UML describes the Artefact class:

<table>
<thead>
<tr>
<th>Artefact</th>
</tr>
</thead>
<tbody>
<tr>
<td>- description : string</td>
</tr>
<tr>
<td>- material : Material</td>
</tr>
<tr>
<td>- age : int</td>
</tr>
<tr>
<td>- value : int</td>
</tr>
<tr>
<td>+ Artefact(d : string, m : Material, a : int, v : int):</td>
</tr>
<tr>
<td>+ ~Artefact():</td>
</tr>
<tr>
<td>+ getDescription(): string</td>
</tr>
<tr>
<td>+ getAge(): int</td>
</tr>
<tr>
<td>+ getValue(): int</td>
</tr>
<tr>
<td>+ getMaterial(): Material</td>
</tr>
<tr>
<td>+ setValue(v: int): void</td>
</tr>
</tbody>
</table>

An artefact is a man-made object of cultural or historical interest. We will describe a single artefact using four member variables:
• Text description (eg. “Crystal Skull”).
• Material (material) the artefact is made of. For the sake of simplicity, there will be only four valid materials: STONE, COPPER, SILVER, GOLD. Note that the variable is of type Material - create a Material enumeration type and store it in Artefact.h.
• age (in years) of the artefact
• Monetary value of the artefact - i.e., the price a museum would be willing to pay for it

Artefact class should have no default constructor. The four-parameter constructor must provide a default value of -1 for the value variable - we may later call an Appraiser to determine artefact’s value. Getter functions must return the values of the member variables. Only one setter function is available - only the value, or price of the artefact can be altered.

Test your class. When you are certain it works as expected, compress Artefact.h and Artefact.cpp into a single archive (either .tar.gz, .tar.bz2 or .zip) and submit it for marking to the appropriate upload link (Assignment 1, Part 1) before the deadline.

3.2 Inventory class (15 marks)

For the Inventory class, create two files: Inventory.h (class definition) and Inventory.cpp (class implementation). The following UML describes the class:

```
Inventory
- inventory: Artefact **
- size: int
- numItems: int

+ Inventory(s : int):
+ ~Inventory():
+ inspectItem(index: int): Artefact *
+ getLastItem(): Artefact *
+ addItem(art : Artefact*): void
+ getNumItems(): int
+ isEmpty(): bool
```

The Inventory class is a re-usable container for the artefacts. It has three private member variables:

• Dynamic 1-dimensional array of dynamically allocated Artefact objects, inventory. Because Artefact objects should also be allocated dynamically, we store pointers to objects in the array.
• size is the maximum size of the inventory. The inventory can be empty (contain no objects), but be of size 10 - i.e., 10 slots for Artefact pointers will be reserved in memory.

• numItems is the number of Artefacts stored in the inventory. When an Inventory object is created, numItems should be set to 0. This number will only increase when artefacts are added to the inventory.

Inventory must provide a default constructor that dynamically allocates the inventory array to 10 elements, unless another size is provided. Destructor must deallocate all Artefact objects stored in inventory, and deallocate the inventory itself. The member functions of Inventory are:

• inspectItem(int) - if there is an Artefact* stored at the provided index, the Artefact* is returned. Otherwise, return NULL and print “Invalid index” to the screen (without the quotes).

• getLastItem() - if the inventory is not empty, return the last artefact and remove it from the inventory. Hint: you do not have to physically remove the artefact address from the inventory - it is enough to decrement the number of artefacts stored in the inventory. If inventory is empty, return NULL and print “Inventory is empty” to the screen (without the quotes).

• addItem(Artefact*) - if there is space in the inventory, store the Artefact* in the first available open position. If the inventory is full, resize the inventory to be double the current size, and then store the Artefact* in the first available open position.

• getNumItems() returns the number of artefacts stored in inventory.

• isEmpty() returns true if the inventory is empty, and false otherwise.

Test your class. When you are certain it works as expected, compress all Inventory and Artefact files (.h and .cpp) into a single archive (either .tar.gz, .tar.bz2 or .zip) and submit it for marking to the appropriate upload link (Assignment 1, Part 2) before the deadline.

3.3 Cave class (5 marks)

Caves may contain ancient artefacts. For the Cave class, create two files: Cave.h (class definition) and Cave.cpp (class implementation). Provided UML describes the Cave class.

Cave has a single member variable: an Inventory object artefacts. Cave has a default constructor that you can leave blank - the artefacts object will automatically invoke the default constructor of Inventory if you have implemented it correctly. Cave has three member functions:

• addArtefact(Artefact *) - adds an artefact to the artefacts inventory.
Cave artefacts: Inventory

+ Cave():
+ ~Cave():
+ addArtefact(a : Artefact *): void
+ hasArtefact(): bool
+ removeArtefact(): Artefact *

- hasArtefact() - returns true if artefacts inventory is not empty, and false otherwise.
- removeArtefact() - returns the last item from the artefacts inventory.

Test your class. When you are certain it works as expected, compress all Cave, Inventory, and Artefact files (.h and .cpp) into a single archive (either .tar.gz, .tar.bz2 or .zip) and submit it for marking to the appropriate upload link (Assignment 1, Part 3) before the deadline.

3.4 Museum class (10 marks)

Museums collect interesting artefacts and offer tours. For the Museum class, create two files: Museum.h (class definition) and Museum.cpp (class implementation). The following UML describes the Museum class:

<table>
<thead>
<tr>
<th>Museum</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name: string</td>
</tr>
<tr>
<td>- budget: int</td>
</tr>
<tr>
<td>- exhibits: Inventory</td>
</tr>
<tr>
<td>+ Museum(n: string, b : int):</td>
</tr>
<tr>
<td>+ ~Museum():</td>
</tr>
<tr>
<td>+ buyArtefact(artefact : Artefact *): int</td>
</tr>
<tr>
<td>+ giveTour(): void</td>
</tr>
</tbody>
</table>

Museum has three private member variables:

- name of the museum, such as “National Gallery”.
- Museum’s budget - amount of money the museum can spend to buy new exhibits (i.e. artefacts).
- exhibits, an Inventory object that the museum uses to manage its exhibits.

Museum has no default constructor, but provides a default value of 1000 for initial budget. Museum has two public functions:

- buyArtefact(Artefact *) - if artefact’s value is positive and within museum’s budget, the artefact is added to exhibits, the budget is decreased
by the value of the artefact, and the amount paid is returned. If artefact’s value is below zero, then the artefact has not been evaluated yet: return 0 and print the following message:

<artefact description> has not been evaluated yet

where <artefact description> must be replaced by the description of the artefact. If artefact’s value is higher than the current budget, return 0 and print the following message:

<artefact description> is out of museum’s budget

where <artefact description> must be replaced by the description of the artefact.

• giveTour() - gives a tour of the museum. First, a welcome message is printed:

Welcome to <museum name>, dear visitor!

where <museum name> is replaced by museum’s name. If there are no exhibits to show, print:

Unfortunately, there is nothing to show.

Otherwise, iterate through the exhibits, displaying the following message for each item:

To the <right, left> you see the <description>, it is <age> years old, and it is made of <stone, copper, silver, gold>.

where <right, left> is replaced by right if the current exhibit’s index is even, and left if the current exhibit’s index is odd. <description> and <age> should be replaced by the description and age of the artefact, respectively. The material of the artefact should be used to print the correct description of artefact’s material (stone, copper, silver, or gold). Whenever a tour is successfully given, add 10 to museum’s budget (tour price).

Test your class. When you are certain it works as expected, compress all Museum, Inventory, and Artefact files (.h and .cpp) into a single archive (either .tar.gz, .tar.bz2 or .zip) and submit it for marking to the appropriate upload link (Assignment 1, Part 4) before the deadline.

3.5 Adventurer class (10 marks)

Adventurers wander through caves in search of artefacts, and then sell the artefacts to museums. For the Adventurer class, create two files: Adventurer.h (class definition) and Adventurer.cpp (class implementation). Provided UML describes the class.
<table>
<thead>
<tr>
<th>Adventurer</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name: string</td>
</tr>
<tr>
<td>- gold: int</td>
</tr>
<tr>
<td>- inventory: Inventory</td>
</tr>
</tbody>
</table>

+ Adventurer(string n):
+ ~Adventurer():
+ takeArtefact(cave : Cave&): void
+ sellArtefact(museum : Museum&): void
+ sellAllArtefacts(museum : Museum&): void
+ getGold(): int

Adventurer has three private member variables:

- **name** of the adventurer, such as “Indiana Jones” or “Lara Croft”.
- **gold** - the amount of money in adventurer’s purse.
- **inventory**, an Inventory object where the adventurer keeps his/her artefacts.

Adventurer’s constructor sets adventurer’s name to the given string, and adventurer’s gold to 0. The following functions are available:

- **takeArtefact(Cave&)** - removes an artefact from the cave and puts it into the inventory.
- **sellArtefact(Museum&)** - if the inventory is not empty, sell the last artefact to a museum. If the museum buys the artefact, the artefact’s value is added to the adventurer’s gold variable.
- **sellAllArtefacts(Museum&)** - attempt to sell all artefacts in the inventory to a museum.
- **getGold()** - returns the total gold of an adventurer.

Test your class. When you are certain it works as expected, compress all of your code (.h and .cpp) into a single archive (either .tar.gz, .tar.bz2 or .zip) and submit it for marking to the appropriate upload link (Assignment 1, Part 5) before the deadline.

### 3.6 Appraiser class (10 marks)

For the Appraiser class, create two files: **Appraiser.h** (class definition) and **Appraiser.cpp** (class implementation). An appraiser is a specialist in ancient artefacts who can estimate the value of an artefact by considering its age and material. The Appraiser has no member variables, and a single member function:

```cpp
void evaluateArtefacts(Adventurer&);
```
The purpose of this function is to go through adventurer’s inventory and evaluate (assign a value) every artefact in the inventory that does not have a valid value yet. If an artefact already has a valid (positive) value, the appraiser should not alter it. `evaluateArtefacts(Adventurer&)` requires direct access to adventurer’s inventory - you will have to modify the Adventurer class to enable such behaviour.

To evaluate a single artefact, first consider its age. If the artefact is older than 100 years, assign a value of 100 to it. If the artefact is older than 50 years, assign a value of 50 to it. If the artefact is older than 10 years, assign a value of 10 to it. Otherwise, start with the value of 1.

Next, consider the material of the artefact. If it is made of gold, multiply the current value by 4. If it is made of silver, multiply the value by 3. If it is made of copper, multiply the value by 2. If it is made of stone, multiply by 1.

When `evaluateArtefacts(Adventurer&)` terminates, all items in adventurer’s inventory should be evaluated.

Test your class. When you are certain it works as expected, compress all of your code (.h and .cpp) into a single archive (either .tar.gz, .tar.bz2 or .zip) and submit it for marking to the appropriate upload link (Assignment 1, Part 6) before the deadline.

The End