Faculty of Science, Engineering and Technology

Object Oriented Programming

Topic 7: Introducing C++

Resources

The following resources can help you with this topic:

- www.cplusplus.com Tutorials and Forums
  - Classes (I)
  - Default constructor and Destructor (stop at copy constructor)
  - Inheritance (skip friendship)
  - Polymorphism
- Tutorials Point C++ Programming Tutorials

Topic Tasks

Before starting to work on the tasks, first read through this entire document to get a sense of the direction in which you are heading. Complete the following tasks and submit your work to Doubtfire for feedback before the deadline (check Doubtfire for the submission deadline).

Supplementary Exercise - C++ Classes and Inheritance

Supplementary Exercise - Planetary Rover UML Class Diagram

Supplementary Exercise - Planetary Rover Code

Credit Task 2 - The Electoral System (Implementation)

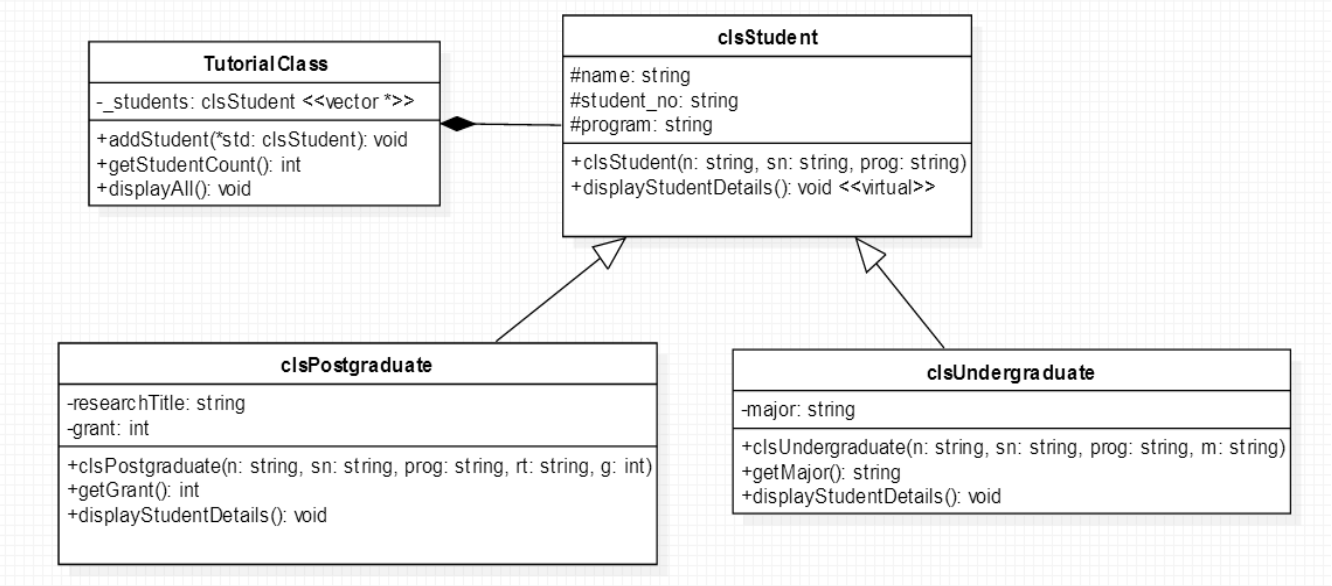
After you have **discussed** your work with your tutor and **corrected any issues**, it will be signed off as complete.

Before starting this exercise, install the C++ IDE, Compile Tools and Unit Testing Library as described in the OOP C++ Setup instructions (available on Blackboard).

SWINBURNE UNIVERSITY OF TECHNOLOGY

Supplementary Exercise: C++ Classes and Inheritance

Given the following class diagram: -



Implement the following:

- Code the classes and methods identified into a C++ program. (Note: Your class file must reflects the attributes and methods given in the class diagram)
- Create unit tests for the following functions:
  - getStudentCount()
  - getGrant()
  - getMajor()
- The code must follow a self-consistent coding convention and be well documented.

Note: this exercise is part of preparation for Pass Task 13

.

Supplementary Exercise: Planetary Rover UML Class Diagram

OOP C++ Setup must be performed accordingly.

For this exercise, you will develop code to model a simple robotic Rover that has a number of types of Device that it can deploy while exploring distant planets.

Use the following description to **draw** a **UML class diagram** of the Planetary Rover model:

★ A Rover has one or more Batteries, which it uses to power its Devices.

★ A Battery has an integer number of units of power (charge) that must be greater than zero.

★ A Rover has a number of Devices, which must be connected to a Battery to operate.

★ Devices can be connected to and disconnected from a Battery.

★ Rover can attach or detach a Device; when a new device is attached, the Rover connects it to the Battery with the greatest charge.

★ Rover can operate; when operated, it simply tells each of its attached devices to operate.

★ There are three different kinds of Device: Radar, Solar Panel, and Drill. When operated, each outputs a unique message and each has a different effect on the Battery to which it is connected

- Radar - when operated, the radar subtracts 4 units of power from the Battery to which it is connected. If successful, it outputs the message “PING" and randomly prints the message “PONG” (e.g. in 1 out of 4 times).
- Solar Panel - when operated, the solar panel adds 1 unit of power to the Battery to which it is connected. If successful, it outputs the message“Charging”.
- Drill - when operated, the drill first checks to see if its safety switch is enabled. If the safety switch is enabled, it prints the message “Safety First” and does nothing more. If the safety switch is not enabled, the drill subtracts 10 units of power from the Battery to which it is connected. If successful, it outputs the message “VRMM! VRMM!”.

Hint: Add a bool field _safetyEnabled to the Drill class. The constructor should initialise it to false. You can then check the behaviour of this switch when writing a test driver.

Note: this exercise is part of preparation for Pass Task 13

Supplementary Exercise: Planetary Rover Code

Create a Rover Tests file and add unit tests for attaching, detaching, and operating devices:

1. Edit **TestRover.h** and replace the two example tests

```
CPPUNIT_TEST(testMethod);
CPPUNIT_TEST(testFailedMethod);

[…]

private:
  void testMethod();
  void testFailedMethod();
```

with the declaration of a single unit test named **testAttach**.

2. Likewise, edit **TestRover.cpp** and replace the two example tests

```
void TestRover::testMethod() {
    CPPUNIT_ASSERT(true);
}

void TestRover::testFailedMethod() {
    CPPUNIT_ASSERT(false);
}
```

with a single **testAttach** method definition

```
void TestRover::testAttach()
{
    Rover* rover = new Rover();
    rover->attachDevice( new Radar() );

    CPPUNIT_ASSERT( rover->deviceCount() == 1 );
    delete rover;
}
```

This is the start of a test class that creates a *Rover* object and tests its *attachDevice* method.

3. Try to build the program; you will get a list of compilation errors related to there being no **Rover** or **Radar** classes. Define the Rover and Radar classes; for each class, use NetBeans to create a new C++ class

   - select **File** -> **New File**.
   - At **Step 1. Choose File Type**
   - under **Categories** select **C++**
   - under **File Types** select **C++ Class**
   - click **Next >**
   - In the **New C++ Class** pop-up window, specify the class name (Rover or Radar) and click **Finish**

4. Note that NetBeans creates both Rover.h (and Radar.h) in the **Header Files** and Rover.cpp (and Radar.cpp) in the **Source Files**.

5. Edit **TestRover.cpp** and, just under the

   ```
   #include "TestRover.h"
   ```

   add `#include` statements to include the two new header files

   ```
   #include "Rover.h"
   #include "Radar.h"
   ```

6. Edit **Rover.h** and **Rover.cpp** and add empty stubs for the **attachDevice** and **deviceCount** methods; e.g. in Rover.h add the *method declaration*

   ```
   int deviceCount () const;
   ```

   (in the public section) and in Rover.cpp add the *method definition*

   ```
   int Rover::deviceCount () const
   {
       return 0;
   }
   ```

Note the use of the `const` keyword in the above declaration and definition.

This tells other developers (and the compiler) that this method does not modify the object that was used to call it. It is a promise that is enforced by the compiler. If any line of code inside the method definition attempted to modify an attribute/field of the class, the compiler would abort with an error message.

Note: Something is intentionally missing in the instructions!

What is the type of argument to attachDevice? Do you need to add another new class?

What are the relationships between this class and the Rover and Device classes?

7. After implementing the missing class, **run** the test again to see it compile and fail; e.g.

```
Test name: TestRover::testAttach
assertion failed
- Expression: rover->deviceCount() == 1

Failures !!!
```

8. To work around this failed test, change the deviceCount method to

```
int Rover::deviceCount () const
{
    return 1;
}
```

   then re-run the test and see it Pass. This indicates that one unit test is not sufficient.

9. Return to the **TestRover** code and add an additional unit test that attaches two devices and ensure that it fails when it asserts that deviceCount == 2.

10. Return to the Rover class and implement the *attachDevice* and *deviceCount* methods.
    - Use the Standard Template Library vector class to add a field called **_devices**
    - In *attachDevice* add the passed pointer to the collection using *push_back*.
    - In *deviceCount* return the size of the collection.

Hint: You will need a vector of pointers, such as

```
vector<Spell*> _spells;
```

(but of course not a collection of spells).

11. Run the unit tests again, and verify that both versions of **testAttach** now pass.

12. Use test-driven development to write unit tests and implement the remaining classes and missing functionality, including
    - Battery, Solar Panel, and Drill classes; and
    - Rover::operate and Device::operate methods
    - Rover::connect and Device::connect (and disconnect) methods for connecting and disconnecting Battery objects

13. Decide how to deal with the error that should arise when a device tries to draw too much charge from a Battery; this occurs when the amount drawn by a Device would reduce the remaining charge to zero (or less). Should the Battery throw an exception or simply return an error code? How would you write a unit test to verify that an exception is thrown when it should be?

Hint: When implementing the random behaviour of the Radar class, use the Standard C Library `rand` and `srand` functions to generate random numbers, as described at http://www.cplusplus.com/reference/cstdlib/rand/

Credit Task 2: The Electoral System (Implementation)

*** Compulsory to student aiming for Credit grade and above**

Implement the ICT Club Electoral System based on UML class diagram you have developed earlier in C++. You are expected to implement the following:

The Classes

- Code the classes and methods identified into a C++ program. (**Note**: Your class file must reflects the attributes and methods given in the class diagram)
- Create at least 5 unit tests for any three classes:
- The code must follow a self-consistent coding convention and be well documented.
- To be signed off, your submission must meet the same criteria as specified.

The Program (Advance Requirements)

Develop the electoral console application with the following functionalities:

- Read club members info from text file (automatically when program started)
- Run Election by election committee
  - Segregate members who are eligible to vote from member list
  - Setup candidates
- Register voter
- Allows voters to vote
- Report Election Result

Credit Task 2 - Assessment Criteria

Make sure that your task has met the following minimum requirements in your submission:

- Classes are all implemented as stated in the design.
- The unit tests correctly check the class implemented.
- The code must follow a self-consistent coding convention and be well documented
- The code must compile and the screenshot must show the program running.