Department of Computer Science
COS110 - Program Design: Introduction
Practical 6

1 Introduction

Deadline: 2nd of October, 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 polymorphism, function templates and exceptions 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. Be sure to submit your files in a .zip or .tar archive. Other file formats are likely to cause problems.

1.5 Mark Distribution

Activity	**Mark**
Function Templates	10
Polymorphism	15
Exception Handling	15
Total	**40**

2 Practical

2.1 Function Templates

Templates are a feature in C++ that enable a function (and as we will see later classes) to accept generic parameters. Rather than coding a function to accept a specific kind of input parameter type, it can be coded to accept a generic type that can then be implemented to respond to different kinds of inputs.

2.2 Exceptions and Polymorphism

Exceptions in C++ are programmatic responses to circumstances that arise during program execution that are problematic in some way. Problematic in the sense that an error has generally occurred. Exceptions and exception handling provide a way for programmers to handle errors during runtime without having to halt program execution.

Polymorphism is a concept that refers to ”many forms”. Specifically in C++, this refers to the ability of a call of a member function of a class to produce different behaviour dependent on the type of the object that invoked it. It is typically seen in instances of an inheritance hierarchy where the derived classes produce different behaviour to their parent through the use of the same functions.

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.

2.3 Task 1

You are working as a programmer designing a vehicular system for the newly forced *Edison Arms Company.* The weapon system is a generic weapon housing that can fit a number of different weapons that are all controlled by a tank in the same way. Most importantly, is the emphasis on safety. During combat, the weapon systems cannot become unreliable or fail lest the pilots be put in unnecessary danger. The weapons generate heat which will also be a factor. Therefore you will need to provide mechanisms to combat this.

2.3.1 fireControl

This is the mounting system for the weapons. It can store a number of weapons and control them as well as handle problems. It is defined as follows:

```
fireControl
-weapons:weapon **
-numWeapons: int
---------------------------
+fireControl(numWeapons:int, weaponList: string *)
+~fireControl()
+accessWeapon(i:int):weapon *
```

The class variables are as follows:

- weapons: A 1D array of weapon pointers. It must be able to accept any type of weapon defined in the hierarchy.
- numWeapons: The number of weapons that are mounted into the system.

The class methods are as follows:

- fireControl: This is the constructor. It will receive a list of weapon names as a string array plus the number of weapons. It must allocate memory for the weapons variable and create a weapon based on the information provided by the string array. Create one weapon of the type indicated at each index of the array. If the name of the weapon contains missile, then a missile type weapon must be created and similarly for laser. This should not be case sensitive. For example given the string [”Laser Beam”, ”laser rifle”,”missile pod”, ”missiles”], 4 weapons would be created, the first two being of the class laser, and the last two of the class missile. The default strength for a laser weapon should be set to 5 when creating laser weapons.
- ∼fireControl: The class destructor. It must deallocate all of the memory assigned to the class.
- accessWeapon: This receives an int which provides an index in the weapons list. It will return the weapon that is stored at that position. If no such weapon is found there, then throw a weaponFailure exception.

2.3.2 weaponFailure

This is a custom exception class used in the context of this system. It will inherit publicly from the exception class. You will need to override specifically the **what** method to return the statement ”Weapon System Failure!” without the quotation marks. The name of this class is **weaponFailure**. This exception will be used to indicate a failure of the weapon system. You will implement this exception in the **fireControl** class as a struct with public access. You will need to research how to specify exceptions due to the potential for a "loose throw specifier error" and what clashes this might have with a compiler. Remember that **implementations** of classes and structs must be done in .cpp files.

As a hint, all of the exception structs in this practical will have three functions:

- A constructor
- a virtual destructor with a throw() specifier
- a const what function which returns a const char* with a throw() specifier.

2.3.3 ammoOut

This is a custom exception class used in the context of this system. It will inherit publicly from the exception class. You will need to override specifically the **what** method to return the statement "Ammo Depleted!" without the quotation marks. The name of this class is **ammoOut**. This exception will be used to indicate a depletion of ammunition for a weapon. You will implement this exception in the **weapon** class as a struct with public access. You will need to research how to specify exceptions due to the potential for a "loose throw specifier error" and what clashes this might have with a compiler. Remember that **implementations** of classes and structs must be done in .cpp files.

2.3.4 Weapon Parent Class

This is the parent class of **laser** and **missile**. Both of these classes inherit publicly from it. It is defined according to the following UML diagram:

```
weapon
-ammo:int
-type:string
-name: string
--------------------
+weapon()
+weapon(a:int, t:string, n:string)
+getAmmo():int
+setAmmo(a:int):void
+getType():string
+setType(s:string):void
+getName():string
+setName(s:string):void
+ventWeapon(heat:T):void
```

+∼weapon()
+fire()=0:string
The class variables are as follows:

- ammo: The amount of ammo stored in the weapon. As it is fired, this will deplete.
- type: The type of the weapon as a string which relates to its class.

The class methods are as follows:

- weapon: The default class constructor.
- weapon(a:int, t:string, n:string): The constructor. It will take three arguments and instantiate the class variables accordingly with name being the last variable set.
- getAmmo/setAmmo: The getter and setter for the ammo.
- getType/setType: The getter and setter for the type.
- getName/setName: The getter and setter for the name.
- ∼weapon: The destructor for the class. It is virtual.
- fire: This is the method that will fire each of the weapons and produce a string of the outcome. It is virtual here.
- ventWeapon: This is a template function. It will receive a generic parameter representing some amount of heat. When called the function should determine the number of cooling cycles needed for the weapon to cool based on the amount of heat that is passed in. For every 10 units of heat, 1 cycle will be needed. You need to display a number of lines of output (with new lines at the end) to represent this. For example, if there's 50.83 heat passed or 50 heat passed in, the output should be:

  ```
  Heat Cycle 1
  Heat Cycle 2
  Heat Cycle 3
  Heat Cycle 4
  Heat Cycle 5
  ```

  Pay attention to the format of the output messages. If the heat is less than 10, then display with a newline at the end:

  ```
  Insufficient heat to vent
  ```

2.3.5 laser

The ionCannon is defined as follows:

```
laser
-strength:int
------------------------------
+laser(s:int)
+~laser()
+setStrength(s:int):void
+getStrength():int
+fire():string
```

The class variables are as follows:

- strength: The strength of the laser. Lasers get stronger the longer they are fired.

The class methods are as follows:

- laser: The class constructor. This receives an initial strength for the laser.
- ∼laser: This is the destructor for the laser. It prints out "X Uninstalled!" without the quotation marks and a new line at the end when the class is deallocated. X refers to the name of the weapon. For example:

  ```
  Tri Laser Cannon Uninstalled!
  ```

- fire: If the laser still has ammo, it must decrease the ammo by 1. It will also increase the strength by 1. It will return the following string: "X fired at strength: Y" where Y represents the strength before firing and X, the name of the weapon. Do not add quotation marks. If ammo is not available, instead throw the **ammoOut** exception.
- getStrength/setStrength: The getter and setter for the strength variable.

2.3.6 missile

The missile is defined as follows:

```
missile
------------------------------
+missile()
+~missile()
+fire():string
```

The class methods are as follows:

- missile: This is the constructor for the class.
- ∼missile: This is the destructor for the missile. It prints out "X Uninstalled!" without the quotation marks and a new line at the end when the class is deallocated. X refers to the name of the weapon. For example:

  ```
  Missile Rack Uninstalled!
  ```

- fire: If the rifle still has ammo, it must decrease the ammo by 1. It will return the following string: "X fired!". X refers to the name of the weapon. Do not add quotation marks. If ammo is not available, instead throw the **ammoOut** exception.

You should use the following libraries for each of the classes:

- fireControl: iostream, string, sstream, algorithm, cstring, exception
- weapon: string, iostream, exception, sstream

You will have a maximum of 10 uploads for this task. Your submission must contain **fireControl.h, fireControl.cpp, laser.h, laser.cpp, missile.h, missile.cpp, weapon.h, weapon.cpp,main.cpp** and a makefile.