UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

# Department of Computer Science
# COS110 - Program Design: Introduction
# Practical 2



## 1 Introduction

### Deadline: 6th of September, 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 polymorphism 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.

### 1.5 Mark Distribution

| **Activity** | **Mark** |
|---|---|
| Classes | 20 |
| Exception Handling | 10 |
| **Total** | **30** |

## 2 Practical

### 2.1 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.

### 2.2 Task 1

You are working as a programmer designing a space ship weapon system for the newly forced *Space Force*. The weapon system is a generic weapon housing that can fit a number of different weapons that are all controlled by a ship pilot 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. Therefore you will need to provide mechanisms to combat this.

#### 2.2.1 weaponMount

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:

```
weaponMount
-weapons:weapon **
-numWeapons: int
--------------------------
+weaponMount(numWeapons:int, weaponList: string *)
+~weaponMount()
+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:

- weaponMount: This is the constructor. It will receive a list of weapons 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. For example ["Laser Cannon Q","Ion Cannon"] means create a laserCannon weapon at index 0 in quad fire mode and so on. "Laser Cannon S" would imply to create a laserCannon weapon with the single firing mode. The default strength for an ion cannon is 5.
- ~weaponMount: 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.

### 2.2.2 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 **weaponMount** 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.

### 2.2.3 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 their corresponding .cpp files.

### 2.2.4 Weapon Parent Class

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

```
weapon
-ammo:int
-type:string
-------------------
+weapon()
+weapon(a:int, t:string)
+getAmmo():int
+getType():string
+setAmmo(s:int):void
+setType(s:string):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. Examples include: "Rail Rifle","Laser Cannon","Doom Cannon", "Missile Launcher" and "Ion Cannon".

The class methods are as follows:

- weapon: The default class constructor.
- weapon(a:int, t:string): The constructor. It will take two arguments and instantiate the class variables accordingly.
- getAmmo,setAmmo: The getter and setter for the ammo.
- getType,setType: The getter and setter for the ammo.
- ∼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.

### 2.2.5 ionCannon

The ionCannon is defined as follows:

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

The class variables are as follows:

- strength: The strength of the ion cannon. Ion cannons get stronger the longer they are fired.

The class methods are as follows:

- ionCannon: The class constructor. This receives an initial strength for the ion cannon.
- ~ionCannon: This is the destructor for the ion cannon. It prints out "Ion Cannon Uninstalled!" without the quotation marks and a new line at the end when the class is deallocated.
- fire: If the cannon still has ammo, it must decrease the ammo by 1. It will also increase the strength by 1. It will return the following string: "Ion Cannon fired at strength: X" where X represents the strength before firing. 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.

### 2.2.6 laserCannon

The laserCannon is defined as follows:

```
laserCannon
-firingMode:char
------------------------------
+laserCannon(f:char)
+~laserCannon()
+fire():string
+getMode():char
+setMode(s:char):void
```

The class methods are as follows:

- laserCannon: This is the constructor for the class. It receives a firing mode as a char. The mode can be: 'Q' or 'S' for quad, single or burst fire respectively.
- ~laserCannon: This is the destructor for the laser cannon. It prints out "Laser Cannon Uninstalled!" without the quotation marks and a new line at the end when the class is deallocated.
- getMode/setMode: The getter and setter for the mode variable.
- fire: If the cannon still has ammo, it must decrease the ammo by the amount determined by the firing mode. You cannot fire if you do not have all the ammo available for the mode. A single laser cannon uses 1 ammo. A quad laser cannon uses 4 ammo. If fired as a single, it will return the following string: "Laser Cannon fired!". If fired as a quad, it will return the following string: "Laser Cannon Quad Burst fired!". Do not add quotation marks. If ammo is not available, instead throw the **ammoOut** exception.

You will be allowed to use the following libraries: **sstream,exception, cstring,string, iostream**. You will have a maximum of 10 uploads for this task. Your submission must contain **weaponMount.h, weaponMount.cpp, ionCannon.h, ionCannon.cpp, laserCannon.h, laserCannon.cpp, weapon.h, weapon.cpp, main.cpp** and a makefile.