1 General Instructions

- This assignment should be completed individually.
- Be ready to upload your assignment well before the deadline. No extension will be granted.
- If your code does not compile you will receive a mark of 0. Marks will primarily be awarded for output, and internal structure may also be tested.
- Read the entire assignment thoroughly before you start coding.
- Follow the specifications exactly, especially with regards to class, function and file names. If anything doesn’t correspond to the specification, it won’t compile on fitchfork, even though it might compile when you test it, because you might be testing it with the wrong name as well.
- To ensure that you did not plagiarize, your code will be inspected using dedicated software. Note that plagiarism is considered a serious offense. 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 several classes that could be used in a game about spaceships. Different types of spaceships are represented by different classes in a hierarchy. Some spaceships have a shield for extra protection, and some are able to attack other spaceships. Each spaceship belongs to a team in a match. Spaceships can be damaged during a game, and when a spaceship is completely destroyed, it removes itself from the match. When one team has no more spaceships, the match is over. An upgrade centre upgrades the stats of spaceships. Different upgrades are available for different types of spaceships.
3 Task [50]

This assignment will give you the opportunity to work with inheritance, polymorphism, abstract base classes, friend functions, and multiple inheritance. You will create 10 classes that interact through their public interfaces, friend functions, and polymorphism. Create the classes as they are discussed in each task below. Test your work for each task thoroughly before submitting to the corresponding GitHub upload slot and moving on to the next task. Each task is evaluated separately, but will rely on the classes you created in previous tasks. New tasks may even require you to modify old classes.

The classes you will create for each task are:

1. SpaceShip, Drone, Match
2. ShieldedSpaceShip, Carrier
3. AttackSpaceShip, StarFighter, Destroyer
4. UpgradeCentre
5. PlanetSinker

3.1 SpaceShip, Drone and Match [10]

The SpaceShip class is an abstract base class that describes the basic traits of all spaceship objects. Create files named `spaceship.h` (for the class declarations) and `spaceship.cpp` (for the class function definitions) and create the SpaceShip class according to these specifications.

The following UML summarises the class members:

<table>
<thead>
<tr>
<th>SpaceShip</th>
</tr>
</thead>
<tbody>
<tr>
<td># hull : double</td>
</tr>
<tr>
<td># match : Match&amp;</td>
</tr>
<tr>
<td>+ SpaceShip(m : Match&amp;, t : Team)</td>
</tr>
<tr>
<td>+ blowUp()</td>
</tr>
<tr>
<td>+ takeAHit(damage : double)</td>
</tr>
<tr>
<td>+ printDescription()</td>
</tr>
</tbody>
</table>

Note that, in UML, traits that are presented in italics are abstract (pure virtual in C++). Implement the class so that its members comply with the following specifications:

- **double hull**
  This member variable represents the hull strength of a spaceship object. When a spaceship is damaged, this value decreases. When this value reaches 0, a spaceship is destroyed.

- **Match &match**
  This member variable is a reference to a Match object. (You will also create the Match class during this task.) It represents the match that this spaceship object partakes in. Note that, because the variable is a reference, it must be initialized during construction.

- **SpaceShip(Match&, Team)**
  The only constructor. The object’s match reference is set to the given Match& parameter, and the spaceship object is added to that match object on the team given as the second parameter. (Team is an enumeration that you will create in Match’s header file.)

- **void blowUp()**
  Represents the event where a spaceship object is destroyed during a match; removes the spaceship object from its match. This function will not be overridden by any derived classes.

- **void takeAHit(double)**
  This function represents a spaceship object being damaged. The parameter represents the amount of damage dealt. The damage is simply subtracted from the hull variable until it reaches 0 (the hull value should never be less than 0). If the resulting hull value is 0, the ship should blow up. Note that some derived classes will override this function; to make sure that those classes exhibit correct polymorphic behaviour, you should declare this function as virtual here.

- **void printDescription()**
  This function is abstract (i.e. pure virtual) and must be overridden by all concrete derived classes. Derived classes will implement this function to print a short description of the object and its properties.
Due to the presence of pure a virtual function, the \texttt{SpaceShip} class is abstract and cannot be instantiated. With inheritance, we can create derived classes of \texttt{SpaceShip} that override the abstract function, and can thus be instantiated. Such classes are then concrete (as opposed to abstract) derived classes. You will create a class called \texttt{Drone} that is a concrete derived class of \texttt{SpaceShip}. Create the \texttt{Drone} class in files called \texttt{drone.h} and \texttt{drone.cpp} according to these specifications:

\begin{table}[h]
\begin{tabular}{|c|}
\hline
\textbf{Drone} \\
+ Drone(Match&, t : Team) \\
+ takeAHit(damage : double) \\
+ printDescription() \\
\hline
\end{tabular}
\end{table}

\texttt{Drone(Match&, Team)}

The constructor performs the same initialisation as the base class’s constructor. In addition, it initialises \texttt{hull} to 20.0.

\texttt{void takeAHit(double)}

This function is overridden so that a drone will instantly blow up whenever it is damaged, regardless of the amount of damage dealt. The \texttt{hull} value should also be set to 0.

\texttt{void printDescription()}

This function is overridden to allow us to instantiate \texttt{Drone} objects - recall that it is pure virtual in the base class. This function should print a message in the following format:

```
Drone {hull: HULL}
```

followed by a new line, with \texttt{HULL} replaced by the object’s \texttt{hull} value.

Lastly, for this task, you must create the \texttt{Match} class. A match object represents a single match between two teams, where each team has a list or an array of spaceships. The number of spaceships that each team has may not be fixed; spaceships can be added or removed freely at runtime. The \texttt{Match} class provides the following public interface:

\begin{table}[h]
\begin{tabular}{|c|}
\hline
\textbf{Match} \\
+ add(s : SpaceShip*, t : Team) \\
+ remove(s : SpaceShip*) \\
+ get(t : Team, i : int) : SpaceShip* \\
+ getSize(t : Team) : int \\
+ gameOver() : bool \\
\hline
\end{tabular}
\end{table}

\texttt{void add(SpaceShip*, Team)}

Adds the given pointer to a spaceship object to the requested team. It is sufficient to store only the pointer (i.e. deep copying is not required). The newly added item should be at the end of the list.

\texttt{void remove(SpaceShip*)}

Searches for a pointer that matches the parameter and removes it from the first team it is found in. If the last spaceship is removed from one of the teams, a message is printed in the following format:

```
Team 1 is out!
```

(or Team 2, depending on which team is out of spaceships), followed by a newline. If the given pointer is in neither of the teams, the following message is printed:

```
No such spaceship!
```

followed by a newline, and an \texttt{InvalidMove} object is thrown as an exception.

\texttt{SpaceShip *get(Team t, int i)}

Returns the pointer to the spaceship stored at the \texttt{i}-th index of the given team, if it is a valid index for that team. If it isn’t, the following message is printed:

```
No such spaceship!
```

followed by a newline, and an \texttt{InvalidMove} object is thrown as an exception.

\texttt{int getSize(Team)}

Returns the number of spaceships that the requested team has left in the match.

\texttt{bool gameOver()}

Returns true if either of the teams have no more spaceships left.
Note that no specification is given detailing how you should implement the storage. It is left up to you to decide, but if you write a constructor, it must be a default constructor. Pro tip: variably-sized lists are easily maintained using `vector` objects.

Create this class in `match.h` and provide implementation in `match.cpp`. In addition, `match.h` must contain a declaration of a class called `InvalidMove`, which is used as an exception. This class can be left empty. `match.h` must also contain a declaration of an enumeration called `Team` which has two enumerated values, namely `ONE` and `TWO`. This enumeration represents the choice of team on which to apply actions of the `Match` class, such as `add`. Take care to name the values exactly as specified.

Once you have completed these classes, you can start testing your code. Create a file with a `main` function and test your code thoroughly - i.e. test every application of every function of every class. The following snippet demonstrates how your classes should be able to interact with one another, but does not necessarily constitute a complete or sufficient test:

```cpp
Match shootingMatch;
Drone d1(shootingMatch, ONE);
Drone d2(shootingMatch, ONE);
// ...
Drone d5(shootingMatch, ONE);
Drone d(shootingMatch, TWO);
while (!shootingMatch.gameOver()) {
    int aimAt;
    cin >> aimAt;
    try {
        shootingMatch.get(ONE, aimAt)->takeAHit(1);
        cout << "Boom!" << endl;
    } catch (InvalidMove) {
        cout << "You missed!" << endl;
    }
}
```

If you are satisfied that your code works, upload an archive (.tar.gz) containing your `spaceship.h`, `spaceship.cpp`, `drone.h`, `drone.cpp`, `match.h`, and `match.cpp` to the upload slot for task 1 of this assignment on the course website.

### 3.2 ShieldedSpaceShip and Carrier [10]

In addition to creating concrete derived classes, the `SpaceShip` class can also be extended by creating abstract derived classes. `ShieldedSpaceShip` will be such an abstract derived class. `ShieldedSpaceShip` will add a shield to the class to deflect some of the damage to the hull, and override the `takeAHit` function to incorporate this shield. Create the `ShieldedSpaceShip` class in the files `shieldedspaceship.h` and `shieldedspaceship.cpp` according to these specifications. It should have the following members:

- **double shield**
  
  Similar to `hull`, this member variable represents the amount of shield that a spaceship has left. Shield is decreased when a spaceship suffers a hit. When a spaceship’s shield is 0, the spaceship does not blow up; the shield simply stops protecting the hull.

- **ShieldedSpaceShip(Match &m, Team t, double h, double s)**
  
  This constructor initialises the object’s `match` reference with the given `Match&` parameter and adds the object to that match, as per usual. It also initialises the object’s `hull` and `shield` variables to `h` and `s`, respectively.

- **void takeAHit(double damage)**
  
  This function is overridden from the base class to act as follows. If `shield` is 0, `damage` is subtracted from `hull` directly. Otherwise, the spaceship calls `shieldHit` with `damage` to deflect some of the damage. `shieldHit` will reduce the value of `shield` and return a value that represents the damage that the shield could not deflect. Only this undeflected damage should be subtracted from `hull`. As per usual, if `hull` reaches a value of 0, the ship blows up.

- **double shieldHit(double damage)**
  
  This function represents a spaceship’s attempt to shield the hull from damage. The parameter represents the amount of damage inflicted on the spaceship. The function will reduce the shield according to an
algorithm determined by each concrete subclass. Any damage that the shield could not deflect is returned. This function should be abstract, forcing subclasses to override it with their own algorithms to determine how much damage could be deflected. This function shouldn’t be public, as it is only used as a helper function in `takeAHit`, but it should be visible to subclasses so that they can override it.

`ShieldedSpaceShip` does not override the pure virtual function `printDescription()`, and moreover, it introduces a new abstract function called `shieldHit`. Clearly, this is an abstract class, even though it is not the top base class in the hierarchy. This is valid in C++. In fact, any class in an inheritance hierarchy can be abstract. To instantiate objects “of” the class, we just have to derive a concrete class from it, just like `Drone` was derived from `SpaceShip`.

For the next part of this task, you must create a class called `Carrier` (in `carrier.h` and `carrier.cpp`) that inherits from `ShieldedSpaceShip`. A carrier is a shielded space ship that is designed to carry cargo. `Carrier` will be a concrete subclass; therefore, it must override all of `ShieldedSpaceShip`’s abstract functions.

double cargoWeight

The class adds a private member variable detailing the weight of an object’s cargo.

`Carrier(Match &m, Team t, double c)`

The constructor handles a `Match&` and `Team` parameter as usual. It also takes a `double` parameter, which is used to initialise the weight of a carrier’s cargo. This weight will not be changed during a program’s execution. `hull` is initialised to 100, and `shield` to 50.

void printDescription()

This function is overridden to print a message in the following format:

`Carrier {hull: HULL, shield: SHIELD, cargo: CARGOWEIGHT}`

followed by a newline, with `HULL`, `SHIELD`, and `CARGOWEIGHT` replaced by the values of the `hull`, `shield`, and `cargoWeight` member variables, respectively.

double shieldHit(double)

This function specifies how much damage is done to a shielded spaceship’s shields, and how much to its hull. Of course, if the shield is depleted (`shield == 0`), all damage goes to the hull. Otherwise, the shield is reduced by only half the damage inflicted on the spaceship, and a quarter of the damage gets through to the hull directly (the hull damage is only returned; recall that this returned value is used by `takeAHit` to perform the damage on the hull).

For example, if 40 damage is done (via `takeAHit`) to a carrier with a hull strength of 100 and shield strength of 50, then, after the attack, the hull strength should be 100-(40/4) = 90, and the shield 50-(40/2) = 30.

Neither the hull nor the shield strength should ever be below 0.

If an attack causes the shield to be depleted, it still deflects damage as per usual. It only stops protecting the hull from the next attack.

The following UML summarises the members of the new classes:

```
+ ShieldedSpaceShip(match : Match&, team : Team, hull : double, shield : double)
+ takeAHit(damage : double)
+ shieldHit(damage : double) : double

+ Carrier(match : Match&, team : Team, cargo : double)
+ printDescription()
+ shieldHit(damage : double) : double

```

Thoroughly test your code with your own main function. Especially check whether your classes exhibit correct polymorphic behaviour - whether each class’s own version of inherited functions are executed, even when objects of those classes are treated polymorphically as their base classes. For example:
Carrier carrier(shootingMatch, ONE, 1000);
SpaceShip *s = &carrier;
ShieldedSpaceShip *ss = &carrier;
s->printDescription();  // is Carrier’s description
ss->printDescription(); // printed for each call?

When you believe you have completed the task correctly, upload an archive containing all of your classes so far (spaceship.h/.cpp, drone.h/.cpp, shieldedspaceship.h/.cpp, carrier.h/.cpp, match.h/.cpp) to the assignment 2 task 2 upload slot.

3.3 AttackSpaceShip, StarFighter and Destroyer [10]

Similarly to how you extended the SpaceShip class to have shields, for this task, you will create a separate extension to enable spaceships to attack other spaceships. You will create an abstract class called AttackSpaceShip that represents any spaceships that can attack. You will then create a concrete subclass called StarFighter, which attacks using lasers. Lastly, you will create a subclass of StarFighter called Destroyer, which is more powerful and also attacks with missiles every few rounds.

In files called attackspaceship.h and attackspaceship.cpp, create the abstract class AttackSpaceShip according to these specifications. The class inherits from SpaceShip (it is not related to ShieldedSpaceShip). It has the following members:

AttackSpaceShip(Match&, Team)
As with SpaceShip, the constructor simply assigns the given match to the object’s match reference and adds the AttackSpaceShip object to the requested team on that match.

void attack(SpaceShip &other)
This function represents an attack on another spaceship. The takeAHit method will be invoked on the other spaceship with a damage value determined by this object. Leave this function abstract; concrete subclasses will determine the implementation specifics depending on their abilities.

void upgradeWeapons()
This function should also be left abstract. The function is used to improve the strength of an object’s weapons, so concrete subclasses must determine its implementation.

Next, create the StarFighter class in the files starfighter.h and starfighter.cpp, complying with the given specifications. StarFighter is a concrete subclass of AttackSpaceShip. It has the following members:

double laserPower
This protected member variable stores the strength of a starfighter’s lasers. This is how much damage a starfighter will inflict on another spaceship when it attacks.

StarFighter(Match&., Team)
The constructor initialises match and adds the new object to a match, as usual. Furthermore, it initialise the starfighter’s laser power to 20, and its hull to 50.

attack
This function overrides the base class function with the same signature so that it attacks another spaceship as follows: the other spaceship’s takeAHit function is invoked with an amount of damage equal to this starfighter’s laser power.

upgradeWeapons
The abstract function in the base class is overridden to increase the object’s laser power by 10. However, the laser power may not exceed 50; if it is already 50, it must simply remain the same.

printDescription
This function overrides the abstract function in SpaceShip. It prints a message in the following format: StarFighter {hull: HULL, laser power: LASERPOWER} followed by a newline, where HULL and LASERPOWER are the values of the hull and laserPower member variables.
Next, in `destroyer.h` and `destroyer.cpp`, create the `Destroyer` class. It should inherit from `StarFighter` (not from `AttackSpaceShip`), and have the following members:

A member variable for storing the amount of damage a missile does

A member variable that will act as an incrementing round-counter

**Destroyer(Match&, Team)**
In addition to the usual match and team initialisation, the spaceship’s laser power must be initialised to the same value as in `StarFighter`, its missile power to 30, and its hull to 80.

**void attack(SpaceShip&)**
This overridden version of `attack` should always use lasers on the other spaceship (just like `StarFighter` does). Furthermore, every third attack should also use missiles - meaning the `takeAHit` function of the other spaceship should be invoked with this object’s missile damage. To keep track of when to shoot missiles, use the new round-counting member variable, modding it by 3 and incrementing.

**void upgradeWeapons()**
This function should perform the same upgrade on the object’s laser power that it did in `StarFighter`. It should also upgrade the missile power by 15. However, the object’s missile power should not be higher than 90. If this function is called on an object whose missile power is already 90, it should simply not be upgraded. (The lasers can still be upgraded if they are not capped yet.)

**printDescription**
Overridden to print a message in the following format:
`Destroyer {hull: HULL, laser power: LASERS, missile damage: MISSILES}` followed by a newline, with HULL, LASERS and MISSILES replaced by the values of the appropriate member variables.

The members of the new classes for task are summarised in the following UML:

```
SpaceShip

AttackSpaceShip
+ AttackSpaceShip(match : Match&, team : Team)
+ attack(other : SpaceShip&)
+ upgradeWeapons()

StarFighter
# laserPower : double
+ StarFighter(match : Match&, team : Team)
+ attack(other : SpaceShip&)
+ upgradeWeapons()
+ printDescription()

Destroyer
# missileStrength : double
# round : int
+ Destroyer(match : Match&, team : Team)
+ attack(other : SpaceShip&)
+ upgradeWeapons()
+ printDescription()
```

Test your code thoroughly. When you are confident that it is correct, upload an archive containing all of the relevant files (`spaceship.h/.cpp`, `attackspaceShip.h/.cpp`, `starfighter.h/.cpp`, `destroyer.h/.cpp`, and `match.h/.cpp`) to the Assignment 2 Task 3 upload slot.

### 3.4 UpgradeCentre [10]

Our `AttackSpaceShip` class allows us to upgrade the weapons on some of the concrete spaceship objects that we can create. We would also like to be able to upgrade other traits of spaceships, namely shields and hulls appropriate spaceships. Create a class called `UpgradeCentre` (note the spelling!) in files `upgradecentre.h` and `upgradecentre.cpp`, with functions to perform these upgrades. Specifically, the `UpgradeCentre` class should have the following public, void-returning member functions:
upgradeHull
  This function should increase the hull of any appropriate spaceship by 10.

upgradeShield
  This function should increase the shield of any appropriate spaceship by 10.

upgradeWeapons
  This function should call the upgradeWeapons member function on any appropriate spaceships.

You must determine, in terms of polymorphism, what the most general (highest in the inheritance hierarchy) class type is that each function can operate on, and make that type the parameter type of the function. For example, the compiler shouldn’t allow you to call upgradeWeapons on a Drone object, but if you create a new subclass that can attack other spaceships, then upgradeWeapons should also work on that new subclass.

You will probably have to change the existing classes to add these functions as friend functions.

Test your solution and upload all your class files so far (spaceship.h/.cpp, drone.h/.cpp, shieldedspaceship.h/.cpp, carrier.h/.cpp, attackspaceship.h/.cpp, destroyer.h/.cpp, match.h/.cpp, and upgradecentre.h/.cpp) as an archive to the Assignment 2 Task 4 upload slot.

3.5 PlanetSinker [10]

For the final task of the assignment, you will create a class called PlanetSinker in the files planetsinker.h and planetsinker.cpp. This class represents a powerful spaceship that has shields and is able to attack other spaceships. The class should use multiple inheritance to get these traits from different classes. It should inherit from ShieldedSpaceShip in order to have a shield, and from Destroyer to inherit that class’s already-defined attack functions.

The class should have the following new or overridden members:

PlanetSinker(Match&, Team)
  The constructor should pass the match and team parameters to the appropriate base classes. It should initialise the object with a hull of 200 and a shield of 200. The missile power variable that it inherits from Destroyer should be initialised to 50, and the laserPower variable that it inherits from StarFighter to 20.

double shieldHit(double)
  The shieldHit function should be overridden to decrease the shield by a third of the damage parameter, and to let a fifth of the damage parameter through to the hull. Other than that, it should work like the inherited version.

void printDescription
  This function should now print a message in the following format:
  PlanetSinker {hull: H, shield: S, laser power: L, missile damage: M}
  followed by a newline, where H, S, L and M are the values of the corresponding member variables.

The takeAHit function should work exactly like it does for the ShieldedSpaceShip class.

You will probably run into some issues due to the so-called diamond problem. PlanetSinker inherits SpaceShip twice: once via ShieldedSpaceShip, and once via Destroyer. When you try to instantiate a PlanetSinker as a SpaceShip, for example

SpaceShip *p = new PlanetSinker(m, TWO);

then the compiler is going to complain about SpaceShip being an ambiguous base class.

To solve the diamond problem, C++ allows classes to be inherited virtually. This tells the compiler to share one instance of the base class among all its derived classes, which means that the base class is no longer inherited twice by the multiply-inheriting class. As a side-effect, the base class becomes a direct base class of all its subclasses, and since your base class can’t have a default compiler (due to the reference variable), you will have to tweak your existing code a bit to ensure the base class is always constructed correctly.

Test your code thoroughly, and then upload all your files (*.h and *.cpp), except your test program(s) and makefile, in an archive to the Assignment 2 Task 5 upload slot.