COP3331 Object-Oriented Software Design

Flex Projects

University of South Florida
Dept. of Computer Science and Engineering
Dr. Tempestt Neal, Instructor
Flex Project: Animal Shelter

To create an Object-Oriented program which allows the user to adopt/surrender their animals with the animal shelter. This program will terminate if the user declares there are no more customers left. The program will verify that it only accepts cats and dogs to the shelter if the user decides to surrender his/her animal.

The parent class named Animal, will include data members Name and Gender (strings), as well as Age (int), and a static member total_animals, that counts the total amount of animals in the shelter. There will be two subclasses (inheriting from the Parent Class), Cat and Dog. BOTH subclasses will use their constructor to set their data members. Cat will contain the data members Hair_length (string, aka hairless, short, long), Color(string), and vaccine (char). Cat will contain the get_shot function if they would like to give the cat a vaccine shot. Dog will contain the data members Weight (int in pounds), Height(int) and species(string). Dog will also contain a Bool operator == that will determine if another dog currently has the same name. This class will also contain a friend function that adds 10 to the weight of the dog. All classes will contain getter functions for their data members.

The Animal class will contain adopt and surrender member functions, which will adjust the vector of animals accordingly (adopted, will be deleted from memory, and if surrendered then will be added to memory). It will also contain a virtual function that displays a description of the animal. The display function will be included in subclasses.

In the main function, the program will read a file containing animals currently in the shop (10 animals). After reading the file, it will iterate through each customer, requesting what action they would like to take. Once completed, it will appropriately leave the program and leave a message to the user.

**Program will Validate that only cats/dogs are accepted, will validate the gender of the animal to be male or female (m/f), and validate for adopting an animal that
MUST exist within the shelter (aka making sure there is animals available to adopt). The program should display appropriate messages for any errors accordingly.

```
Animal

- name: string
- age: int
- gender: string
- vector<Animal> animals
- static total_animals: int

Animal(name, age, gender): void
+ get_name(): string
+ get_age(): int
+ get_gender(): string
+ adopt(Animal animal_temp): vector<Animal>
+ surrender(Animal animal_temp): vector<Animal>
+ virtual display_animal(): string

Cat

- hair_texture: string
- color: string
- vaccine: char

+Cat(name, age, gender, hair, color): void
+ get_hair_length(): string
+ get_color(): string
+ get_vaccine(): char
+ display_animal(): string
+ get_shot(): Cat

Dog

- weight: int
- height: int
- species: string

+Dog(name, age, gender, weight, height): void
+ get_weight(): int
+ get_height(): int
+ get_species(): string
+ display_animal(): string
+ Bool operator ==(): void
+ friend adjust_weight(Dog): Dog
```
The concept for my flex project is a simple blackjack game. The program will begin by reading from a text file that will contain the objective of the game, scoring, and how to play. The information will then be displayed in the console. The Cards class will create a card object that is assigned a suit, rank, value and string representation. The set_suit() member function, will store “spades”, “clubs”, “hearts” or “diamonds” in the suit variable. The set_rank() member function, will set the rank of the card to a number 2 – 10 or “jack”, “queen”, “king”, or “ace”. The set_value() member function will use the rank of the card to determine how many points the card is worth and store the value in the value variable. The set_card_name() function will use the suit and rank to assign the card object a string representation and store it in the card_name variable. The get_value() function will return the value of the card. The get_card_name() function will return the string representation of the card.

The person class will have four protected data members that will be inherited by the dealer class and the player class. Each person will have a hand, created from a vector of card objects, a current score, a final score, and a number of wins. The games_played data member and get_games_played() member function are static and will be inherited and used by the dealer class. The > and == operators will be overloaded to compare the final scores of the dealer and player in the check_for_win() member function. The += operator will be overloaded and used to add cards to each players hand when the deal_cards() function or hit() function is called from the Dealer class. The take_turn() function will start by checking to see if the player has blackjack. If the player has blackjack, the display_current_stats() function will be called and the final score will be returned, ending the player’s turn. If the player does not have blackjack, the display_current_stats() function will be called, and the player will be presented the option to hit or stay. If the player chooses hit, the hit() function from the Dealer class will be called and a card will be added to the player’s hand and the current score will be updated. This will continue until the player chooses stay. At this point, the turn will end and the final score will be returned. The clear_hand() function, will be used at the beginning of each new game to clear the player’s current hand of all cards so the person can receive new cards. The display_current_stats() function will display the person’s current hand and score. The add_win() function will be used by the check_for_win() for function to update the person’s number of wins after each hand. The calculate_current_score() function will use the default += operator to add the value data members of each card in the person’s hand to the current_score data member. The get_final_score() function will return the value of the current score after the turn is complete. The get_num_wins will return the number of wins for person after each hand. The check_for_win() function will be a friend function that will accept two person objects as parameters. It will use the > and == operators to compare the final scores of each person. Whichever person’s final score is the closest to 21 without going over will be the winner and the result will be displayed on the console. It will be called from the main function after the player and the dealer have taken their turns. If both players have the same final score the game will be a draw, and the result will be displayed in the console. All of the overloaded operators will be inherited by the dealer and player class.

The dealer class will inherit the person class and use object composition to create a deck of cards that will be stored in a vector of Card objects. The create_deck() function will start by clearing the current deck of any cards that are stored in it. It will then increment the static games_played member function by 1. It will use a nested for loop to create the deck of cards.
The first for loop will use the set_suit() function to store the suit of each card. The inner for loop
will create 13 cards for each suit. For each card, the set_rank() function and set_value()
function will be called to assign a rank and value. After each card has been assigned a suit, rank,
and value, the set_card_name() function will be called to assign each card a string
representation. The card will then be added to the deck. The remove_card() function will be
used to remove the first card from the deck after it has been dealt to a person. The
shuffle_deck() function will use the shuffle algorithm to randomly shuffle the deck at the
beginning of each game. The hit() function will accept a person object as a parameter and will
add a single card to the person’s hand and use the remove_card function to remove the card
that has been dealt from the deck. The get_deck() function will return the deck of cards. The
take_turn() function will override the take_turn() function from the Person class, and the dealer
will be required to use the hit function as long as the current score is less than 16. If the score is
greater than or equal to 16, the dealer will be required to stay. At this point the turn will end
and the final score will be returned. The deal_cards() function will be a friend function that
accepts a dealer and a player object as parameters. It will begin by clearing each player’s hand
of any cards by using the clear_hand function. It will the use a for loop and the overloaded +=
operator to add 2 cards to each person’s hand, while removing each card from the deck after it
has been added to a player’s hand.

The player class will also inherit the Person class. The player class will have no data
members except the ones inherited from the Person class. It will use polymorphism to use the
default take_turn() function in the Person class. Unlike the dealer class, the Player class will be
given the option to hit or stay, and the turn will continue until stay is chosen. Once the Player
turn is over, the Dealer turn will begin and run automatically.

I have implemented a scoreboard the will display the current score for each player at
the beginning of each person’s turn and final score at the end of the game. Under the score of
each player, the number of wins for each player will also be displayed. This will be updated at
the end of each hand and the results will be displayed when the scoreboard is shown for the
last time.
**Encapsulation**

Cards.h – line 10  
Dealer.h – line 13

**Inheritance**

Dealer.h – line 11  
Player.h – line 7

**Polymorphism**

Person.h – line 26  
Dealer.h – line 23  
Main.cpp – line 20, 101, 111

**Static Data Members and Functions**

Person.h – line 19, 37  
Dealer.cpp – line 16, 30  
Main.cpp – line 96

**Friend Functions**

Person.h – line 35  
Dealer.h – line 23  
Main.cpp – line 94, 105, 112

**Overload Operators**

Person.h – line 23, 24, 25  
Person.cpp – line 126, 131, 133  
Dealer.cpp – line 98, 151, 153
There are several links to C++ blackjack games, however my program will be unique from each one in multiple ways.

https://codereview.stackexchange.com/questions/133489/c-blackjack-game

This example only uses one class for whole program. My program will use four classes; a class to create cards, a class to create a person, a two subclasses of the person class for the dealer and the player. This example also uses built in arrays to store each players hand. My program will use a vector of card objects defined in the person class for each hand.


This example is more advanced than the last as it uses 6 classes. This example uses a separate class for the deck of cards. In my program, the deck of cards is a member of the dealer class. The dealer class will be responsible for creating the deck of cards and any changes to the deck of cards such as shuffling and dealing cards. My program will also use inheritance to use overloaded operators defined in the person class to compare the scores of the dealer and the player, add the total score of each players hand, and add cards to each players hand.

https://www.dreamincode.net/forums/topic/128198-a-game-of-twenty-one/

https://codereview.stackexchange.com/questions/113816/21-game-similar-to-blackjack


http://www.cplusplus.com/forum/beginner/162557/


All of these examples have little to no classes or are they are incomplete.

https://codereview.stackexchange.com/questions/78710/oop-blackjack-in-c

This example uses a separate class for the deck of cards whereas my deck of cards is a vector of card objects in my dealer class. There are also no instances of polymorphism or overloaded operators in this example.
Description

A fighting game consisting of two champions with specific attacks and a battle to the death!

Player One will be the MeleeChampion. The MeleeChampion has Energy which is used to calculate the strength of the attack and whether the champion can attack that round. Player Two will be the MageChampion. Similar to MeleeChampion, the MageChampion has Mana which determines the strength of the attack and whether the champion can attack that round. Both champions have Health, Defense, and Damage stats. The Health stat is a fixed number at the start of the fight that will go down after each successful attack from the opponent. The Defense stat is a fixed number which determines how much of the opponent’s attack can be blocked or suppressed. Lastly, the Damage stat is what is used in combination with the opponent’s defense stat, energy/mana stat, and a sprinkle of RNG to calculate a final Hit Point number to remove from the defending champion’s health. First to 0 Health loses.

Champion

The Champion class is the main class that keeps track of the name, health, damage and defense. This class will not be allowed to be created on its own. A subclass must extend this class to create an object.

name: string – name of the champion

health: int – a value that goes down as the champion is attacked. 0 means dead!

damage: int – a value initially dealt to an enemy champion.
Flex Project Submission 1

defense: int – a value that weakens an enemy’s damage dealt.

get_name(): string – returns the name of the champion.

get_health(): int – returns the health of the champion.

get_damage(): int – returns the default damage of the champion.

get_defense(): int – returns the defense of the champion.

attack(Champion, int damage): void – attacks a champion, calculating the attacking champion’s damage and defending champion’s defense and coming up with a final hit-point number to take off.

hurt(int amount): void – removes health from champion.

heal(int amount): void – adds health to champion.

MeleeCampion

A melee champion is just a type of Champion that uses physical combat to attack an opponent.

energy: int – value for allowing the champion to attack.

get_energy(): int – returns the value of energy.

attacked(int): void – takes in the damage taken and recalculates health.

head_butt_kick(Champion): void – calculates how much damage to give to opponent.

punch_health_potion(): void – champion punches a health potion to regain some health.
MageChampion

A mage champion is just a type of Champion that uses magic combat to attack an opponent.

**mana**: int – value for allowing the champion to attack.

**get_mana()**: int – returns the value of energy.

**attacked(int)**: void – takes in the damage taken and recalculates health.

**super_cool_magic_spell(Champion)**: void – calculates how much damage to give to opponent.

**punch_health_potion()**: void – champion uses a healing spell to gain some health.

Scoreboard

This is a static class that keeps track of the score, determines if there is a winner, and prints out the scores.

**player1Score**: int – stores the score for player 1.

**player2Score**: int – stores the score for player 2.

**static get_player1Score()**: int – returns the score for player 1.

**static get_player2Score()**: int – returns the score for player 2.

**save_scores()**: void – stores the current scores in a text file.
get_scores(): void – reads the scores from a text file.

set_scores(int, int): sets the scores for the players if reading scores from a text file.

C++ elements locations:

Encapsulation: Champion.cpp, MeleeChampion.cpp, MageChampion.cpp, etc.
Polymorphism: Champion.cpp, line 12.
Static data members and functions: Scoreboard.cpp
Friend functions: None
Overload operators: None

No Extra Credit Applicable****
Flex Project Submission 1
Project: “Monster Arena Game”

Brief Program Description:

The Monster Arena Game allows the player to play as a warrior that fights against three classes of monsters: Wizard, Skeletons, and Ogres. To combat these monsters, the player can use a number of commands: Attack, Magic, Guard, or Use Item. As the player fights and defeats each monster, the player can find an item drop that can be useful for future rounds. The player is also able to choose to increase their health, mana, or strength after defeating an enemy. However, as the player defeats more and more monsters, the monsters scale up and become more challenging with each defeated monster.

The game starts with the player selecting which difficulty they would like to play in: Easy, Medium, or Hard. The difficulty sets the number of rounds the player must endure. For example, Easy mode has the player fight in 5 rounds, Medium is 10 rounds, and Hard is 15. To win the game, the player must survive each round and defeat the monster. On the last round, they fight the boss which is a monster of one of the three types whose difficulty is scaled upwards. However, if the player is defeated before reaching the end, they must restart from the beginning. The game will also keep track of stats of the player (lifetime monsters defeated, games won, etc.) throughout the game. If the player is able to beat the game in the Hard difficulty mode, they will unlock Hardcore mode, which has the player survive 25 rounds.

Concepts Used:

- Encapsulation - Program uses encapsulation in making various data members and functions public, protected, or private.
- Inheritance - The Wizard, Skeleton, and Ogre classes inherit the Monster class and allows these classes to use various member functions and data members.
- Polymorphism - The classes use polymorphism for the monster_intro() function. Each function will override the monster_intro() function from the Monster class to create a string that the game will display when the certain type of monster enters the game.
- Static data members and functions - The program uses several static data members. The Wizard, Skeleton, and Ogre classes each have a static int member that counts when the respective monster is defeated. The Monster class also has a static int member that counts total number of monsters defeated and provides a static function that obtains the total of monsters defeated.
- Friend functions - A friend function is made between the wizard and skeleton that increases the base damage of the wizard from the base damage of the skeleton.
- Overload operators - The insertion operator is used within the Ogre class to display a warning message for when the ogre will throw the boulder.
- **Encapsulation**
  - Monster.h (Lines 7-26)
  - Ogre.h (Lines 9-25)
  - Player.h (Lines 7-24)
  - Skeleton.h (Lines 11-26)
  - Wizard.h (Lines 11-29)

- **Inheritance**
  - Ogre.h (Lines 8-17)
  - Ogre.cpp (Lines 5-9)
  - Skeleton.h (Lines 10-18)
  - Skeleton.cpp (Lines 4-8)
  - Wizard.h (Lines 10-19)
  - Wizard.cpp (Lines 6-10)

- **Polymorphism**
  - Monster.h (Line 23)
  - Monster.cpp (Lines 10-12)
  - Ogre.h (Line 25)
  - Ogre.cpp (Lines 41-42)
  - Skeleton.h (Line 26)
  - Skeleton.cpp (Lines 35-36)
  - Wizard.h (Line 29)
  - Wizard.cpp (Lines 51-52)
  - MonsterDuels.cpp (Lines 437-443)

- **Static data members and functions**
  - Monster.h (Line 14 and Lines 25-26)
  - Ogre.h (Line 14-15)
  - Skeleton.h (Line 16)
  - Wizard.h (Line 17)
  - MonsterDuels.cpp (Lines 20-23)

- **Friend functions**
  - Skeleton.h (Line 25)
  - Wizard.h (Line 28)
  - Wizard.cpp (Lines 45-48)
  - MonsterDuels.cpp (Line 344)

- **Overload operators**
  - Ogre.h (Line 24)
  - Ogre.cpp (Line 36-38)
  - MonsterDuels.cpp (Line 384)
Monster Type Information:

- **Wizards** - Wizards utilize magic attacks in battle. They use three kinds of attacks: normal attack, freeze spell, and explosive spell. The normal attack uses the base damage that is determined by the round the player is in. Explosive attack can deal three times the base damage to the player, but has a 50/50 chance of missing the player. Freeze attack deals only 25% base damage to the player but it also has a 40% chance of freezing the player for one turn. At later rounds, the wizard also has a special ability in which they can summon a skeleton. This allows them to add 20% damage to their base damage for normal attacks.

- **Skeletons** - Skeletons' three different attacks are normal arrow, arrow flurry, and curse arrow. Normal arrows deal the base damage amount. Arrow flurry allows a skeleton to shoot up to 3-6 arrows in a single turn. However, these arrows only amount to 25% of the base damage. Curse arrow only deals 30% base damage. However, it inflicts a curse status on the player that damages them for 10% base damage each turn.

- **Ogres** - Ogres are major damage dealers that typically have high health. Their normal attack deals that base damage. Their heavy club attack deals heavy damage which is 2 times the base damage dealt. After a heavy club attack, an ogre must rest for one turn. Ogres’ most dangerous attack is the boulder throw. This move takes three turns to complete. However, if the third turn is reached, the player takes 4 times the base damage. The player can stop the ogre from doing this attack if the player does any damage to the ogre. Each damage attack from the player has a 30% chance of stopping the boulder throw.

**UML Diagrams**

[Diagram showing classes and methods for Monster, Wizard, Skeleton, and Ogre]
Class Overview

1. Monster

   a. Data Members
      i. health (int) - Protected data member will hold the health data member for
         the monster that is calculated based on the round the game is on.
      ii. base_damage (int) - Holds the base damage that is used for damage
          calculation for all the monsters. Is based on the round the game is on. Is a
          protected data member.
      iii. monster_type (String) - Holds what is the monster’s type (“Wizard”,
          “Skeleton”, “Ogre”)
      iv. is_boss (bool) - Determines if the monster is the final boss of the game.
      v. defeated_monsters (int) - Static data member that holds how many
         monsters were defeated in the game.

   b. Member Functions
      i. Monster(int, int, bool) - Constructor that initializes the health,
         base_damage, and is_boss data members.
      ii. monster_intro() (String) - Function that makes a string that introduces
         the monster. Function will be overridden by subclasses.
      iii. set_health(int) (void) - Sets health data member
      iv. set_base_damage(int) (void) - Sets base_damage
      v. set_monster_type(String) (void) - Sets monster_type
      vi. get_health() (int) - Returns health
      vii. get_base_damage() (int) - Returns base_damage
      viii. get_monster_type() (String) - Returns monster_type
      ix. get_defeated_monsters() (int) - Calculates defeated_monsters and
          returns it
2. **Wizard**
   a. **Data Members**
      i. `explosive_atk (int)` - Holds value of damage given when using an explosive spell.
      ii. `freeze_atk (int)` - Holds value of damage given when using freeze spell.
      iii. `player_frozen (bool)` - Denotes whether the player is frozen or not.
      iv. `skeleton_summoned (bool)` - Denotes whether a skeleton has been summoned.
      v. `defeated_wizards (int)` - Static data member that counts how many wizards have been defeated
   b. **Member Functions**
      i. `Wizard(int, int, bool, String)` - Constructor that uses parameters for health, base_damage, is_boss, and monster_type that helps initialize the data members for the Wizard object. Monster_type will be “Wizard”.
      ii. `set_explosive_atk() : void` - Sets explosive_atk damage value
      iii. `set_freeze_atk() : void` - Sets freeze_atk damage value
      iv. `set_player_frozen(bool) : void` - Sets whether player is frozen or not
      v. `set_skeleton_summon(bool) : void` - Sets whether skeleton has been summoned
      vi. `get_explosive_atk() : int` - Returns explosive_atk
      vii. `get_freeze_atk() : int` - Returns freeze_atk
      viii. `get_player_frozen() : bool` - Returns player_frozen
      ix. `get_skeleton_summon() : bool` - Returns skeleton_summoned
      x. `summon_skel(Wizard, Skeleton) : friend void` - Friend function that adds to the base damage of the wizard. The damage added to the wizard’s base damage is calculated from 20% of the skeleton’s base damage.
     xi. `monster_intro() : String` - Produces a string to introduce the Wizard monster.

3. **Skeleton**
   a. **Data Members**
      i. `num_of_arrows (int)` - Holds number of arrows that will be fired in arrow flurry attack
      ii. `curse_arrow_dmg (int)` - Holds value of damage for curse arrow attack
      iii. `player_cursed (bool)` - Denotes whether the player is cursed or not
      iv. `defeated_skeletons (int)` - Static data member that counts how many skeletons have been defeated
   b. **Member Functions**
      i. `Skeleton(int, int, bool, String)` - Constructor that uses parameters for health, base_damage, is_boss, and monster_type that helps initialize the data members for the Skeleton object. Monster_type will be “Skeleton”.
      iii. `set_curse_arrow_dmg() : void` - Sets curse_arrow_dmg
      iv. `set_player_cursed(bool) : void` - Sets player_cursed
v. get_num_of_arrows() : int - Returns num_of_arrows
vi. get_curse_arrow_dmg() : int - Returns curse_arrow_dmg
vii. get_player_cursed() : bool - Returns player_cursed
viii. summon_skel(Wizard, Skeleton) : friend void - Friend function that adds to the base damage of the wizard. The damage added to the wizard’s base damage is calculated from 20% of the skeleton’s base damage.
x. monster_intro() : String - Produces a string to introduce the Skeleton monster.

4. Ogre
   a. Data Members
      i. heavy_atk_dmg (int) - Holds the damage value for a heavy club attack
      ii. throw_rock_dmg (int) - Holds the damage value for the boulder throw attack
      iii. rock_charge (bool) - Denotes whether the ogre is in sequence for doing a boulder throw.
      iv. charge_rounds (int) - Static data member that counts the number of rounds the ogre charges up/rests.
      v. defeated_ogres (int) - Static data member that counts the number of ogres defeated.
   b. Member Functions
      i. Ogre(int, int, bool, String) - Constructor that uses parameters for heath, base_damage, is_boss, and monster_type that helps initialize the data members for the Ogre object. Monster_type will be “Ogre”.
      ii. set_heavy_atk_dmg() : void - Sets heavy_atk_dmg
      iii. set_throw_rock_dmg() : void - Sets throw_rock_dmg
      iv. set_rock_charge(bool) : void - Sets rock_charge
      v. get_heavy_atk_dmg() : int - Returns heavy_atk_dmg
      vi. get_throw_rock_dmg() : int - Returns throw_rock_dmg
      vii. get_rock_charge() : bool - Returns rock_charge
      viii. operator<<(ostream&, Ogre) : friend ostream& - Overload of the insertion operator that will be used to display warning messages for the rock throw sequence attack
      ix. monster_intro() : String - Produces a string to introduce the Ogre monster.

5. Player
   a. Data Members
      i. health (int) - Holds the max health value of the player
      ii. strength (int) - Holds the strength value of the player
      iii. mana (int) - Holds the max mana value of the player
      iv. guard_stat (double) - Holds the value in which the guard command negates a percentage of damage from the monster
      v. items_held (int) - Holds the number of items held for player
   b. Member Functions
i. **Player()** - Constructor that is used to initialize the base stats of the player at the beginning of the game.

ii. **set_health(int)** : void - Sets health

iii. **set_strength(int)** : void - Sets strength

iv. **set_mana(int)** : void - Sets mana

v. **set_guard_stat(double)** : void - Sets guard_stat

vi. **set_items_held(int)** : void - Sets items_held

vii. **get_health()** : int - Returns health

viii. **get_strength()** : int - Returns strength

ix. **get_mana()** : int - Returns mana

x. **get_guard_stat()** : double - Returns guard_stat

xi. **get_items_held()** : int - Returns items_held

**Player Class Note**

The player class will be working directly with a text file in order to keep everything in check and to read or write values when necessary.

**Additional Notes**

- The program will be reading and writing to text files. This concept will be implemented in the inventory system for the program.

- This project most likely falls into the EC2 category for extra credit opportunities. The whole program is a game where the player must compete with the computer (monsters) and the game’s “scoreboard” will be the stats that the program keeps track of in every playthrough of the game.
## Flex Project Description File

### Title of the Program : Housing Mortgage Analyzer

```
#name : string
#house_price : double
#property_tax : double
#monthly_debt : double
#down_payment : double
#credit_score : int
#worker_count : static int

Worker
+
Worker(name_ : string, house_price_ : double, property_tax_ : double, monthly_debt_ : double, down_payment_ : double, credit_score_ : int)
+ set_name(name : string) : void
+ get_name() : string
+ set_house_price(house_price : double) : void
+ get_house_price() : double
+ set_property_tax(property_tax : double) : void
+ get_property_tax() : double
+ set_monthly_debt(monthly_debt : double) : void
+ get_monthly_debt() : double
+ set_down_payment(down_payment : double) : void
+ get_down_payment() : double
+ set_credit_score(credit_score : int) : void
+ get_credit_score() : int
+ get_worker_count() : static int
+ get_description() : virtual string
```

```
SelfEmployed
-
taxable_annual_salary : double

SelfEmployed
+
SelfEmployed(name_ : string, house_price_ : double, property_tax_ : double, monthly_debt_ : double, down_payment_ : double, credit_score_ : int, taxable_annual_salary_ : double)
+ set_taxable_annual_salary(taxable_annual_salary : double) : void
+ get_taxable_annual_salary() : double
+ get_description() : string
+ operator+(right : const SelfEmployed&) : SelfEmployed
+ combine(employee : Employee&, selfemployed : SelfEmployed&) : friend void
```

```
Employee
-
monthly_salary : double

Employee
+
Employee(name_ : string, house_price_ : double, property_tax_ : double, monthly_debt_ : double, down_payment_ : double, credit_score_ : int, monthly_salary_ : double)
+ set_monthly_salary(monthly_salary : double) : void
+ get_monthly_salary() : double
+ get_description() : string
+ combine(employee : Employee&, selfemployed : SelfEmployed&) : friend void
+ operator+(right : const Employee&) : Employee
```

### Classes Descriptions

1. **Worker Class**

   Worker class is the superclass in this program. This class includes the most important data members to use across its subclasses through inheritance. This class also includes static data member of `get_worker_count` as well as `get_description` to use polymorphism across its subclasses.
• Data Members:

name : name is a protected string data member in the Worker class that used to store the user’s name after prompted user to enter in the main.cpp file. name is used to generate report in the end of the program.

house_price : house_price is a protected double data member in the Worker class that used to store the house’s price that user intend to buy. The price is used to calculate the chance of approval of a conventional mortgage along with some other data members.

property_tax : property_tax is a protected double data member in the Worker class that used to store the annual property tax due of the house that user intend to buy. The property_tax is used to calculate the chance of the mortgage approval along with monthly_debt and other data members.

monthly_debt : monthly_debt is a protected double data member in the Worker class that used to store the monthly debt that the user has. It is used to calculate the chance of approval of a conventional mortgage along with other data members.

down_payment : down_payment is a protected double data member in the Worker class that used to store the down payment for the house purchase that the user intend to put down. It is used with house price along with other data members to calculate the chance of approval of a conventional mortgage.

credit_score : credit_score is a protected int data member in the Worker class that used to store the recent credit score of a worker. It would later be used to determine the chance of approval of a conventional mortgage.

worker_count : worker_count is a protected static int data member in the Worker class that used to count the number of workers that the user intend to add as co-signers for the conventional mortgage. It would be used in the final report of the mortgage approval rate. Since worker_count is state data member, it won’t need an object to call its function. worker_count and its member function is belong to the Worker class.

• Member Functions:

The constructor Worker(name_ : string, house_price_ : double, property_tax_ : double, monthly_debt_ : double, down_payment_ : double, credit_score_ : int) is public and it would be used to store the values of the data members in order to use in its subclasses. These data members would be used to determine the chance of approval of the conventional mortgage.

set_name(name : string) is a public void member function that would be used to set name data member based on the input parameter which would be obtained from the console in the main.cpp file.
get_name() is a public string member function that would be used to output what stored in the name data member as string. It would be used in the main.cpp file to output the name of the user to the console as well as output to the final report txt file.

set_house_price(house_price : double) is a public void member function that would be used to set the house_price data member based on the input parameter that would be obtained from the console in the main.cpp file when prompt the user to enter the house price.

get_house_price() is a public double member function that would be used to output the house price as a double data type. It would be used in the main.cpp to calculate with other members to determine the chance of the approval of the conventional mortgage as well as used to output to the final txt report file.

set_property_tax(property_tax : double) is a public void member function that would be used to set the value of property tax based on the input parameter that would be obtained from the console in main.cpp file.

get_property_tax() is a public double member function that would be used to output the value of property tax to the console and the final txt report file as well as used to calculate with other data member to determine the chance of the approval of the conventional mortgage.

set_monthly_debt(monthly_debt : double) is a public void member function that would be used to set the monthly debt of the user based on the input parameter that get from the console when main.cpp file prompt the user to enter the monthly debt.

get_monthly_debt() is a public double member function that would be used to output the value of monthly debt as a double value. It would be used to calculate with other data member to determine the chance of approval of a conventional mortgage. It would also be used to output to the final txt report file.

set_down_payment(down_payment : double) is a public void member function that would be used to set the value of the down payment that the user agrees to put down for the house purchase based on the input parameter that get from the console when prompting the user to input the value.

get_down_payment() is a public double member function that would output the value that stored in the down_payment data member. It would be used to calculate with other data members to determine the chance of the approval of the conventional mortgage as well as used to output the down payment amount to the final report txt file.

set_credit_score(credit_score : int) is a public void member function that would be used to set the credit score of a worker based on the input parameter that get from the console when asking the user’s input.
get_credit_score() is a public int member function that would output the value that stored in the credit_score data member. It would be used later to determine the chance of the approval of a conventional mortgage as well as displaying to the final txt report file.

get_worker_count() is a public static int member function that would be stored with the Worker class. This member function would be accessed with object of the class because it is static function. This member function would output the number of the workers that the user input. The total worker count would be number of purchaser with cosigner of the conventional mortgage. This value would be used to determine the chance of approval with other data members as well as output to the final txt report file.

get_description() is a public string member function that would also be included in its subclasses in order to use polymorphism to determine which definition of the function that would be used. For the Worker superclass, get_description would include the definition such as worker’s name, house price, property tax, monthly debt and down payment that stored in the object of the Worker class.

2. Employee Class

Employee class is a subclass of the superclass Worker. The Employee class would inherit from the Worker class and add another data member monthly_salary. The Employee class would have a constructor to store the data members and additional member function for setting and getting the monthly_salary data member. Employee class would have a different definition for the get_description() member function so that the program can use polymorphism to determine which specific one to use. This class also has combine friend function that is able to access data members in both Employee and SelfEmployed class. Another member function is the overload + operator to add data members of two of the object of the Employee class together.

- Data Member:

monthly_salary is a private double data member in the Employee class. Monthly salary would be used to store the value of the monthly_salary of an employee based on the user’s input. The console would ask the user if he/she is employee or self-employed. If the user choose employee, the object of employee would be called and the value of the monthly salary would be stored in the object of this class.

- Member Functions:

The constructor Employee(name_: string, house_price_: double, property_tax_: double, monthly_debt_: double, down_payment_: double, credit_score_: int, monthly_salary_: double) is public and it would be used to store the values of the data members in an object of this class. These data members would be used to determine the chance of approval of the conventional mortgage.

set_monthly_salary(monthly_salary: double) is a public void member function that would be used to set the value of the monthly salary, so that it can be used later to calculate with other data members to determine the chance the approval of the conventional mortgage.
get_monthly_salary() is a public double member function that would output the value of monthly salary stored as double data type. It would be used with other data members to calculate the chance of approval as well as it would be output to the final txt report.

get_description() is a public string member function that has the same name and type as the member function in the Worker superclass. The member function has a different definition other than the one in the Worker class. The definition would include such as worker’s name, house price, property tax, monthly debt, down payment and monthly salary of the employee. The program would use polymorphism to determine which definition of the function to call.

combine(employee : Employee&, selfemployed : SelfEmployed&) is a public friend void member function which is able to access the data members from both Employee and SelfEmployed class. In the class header file, I would have forward declaration of the SelfEmployed class as the requirement of the friend function. The friend function then would have a separate implementation file which has definition to convert annual salary of the self-employed worker to monthly salary and add together to employee’s monthly salary for the calculation of the chance of the approval. Because the mortgage company would look the combined monthly salary of the signer and co-signer to determine the debt to income ratio monthly and calculate the chance of approval.

operator+(right : const Employee&) : Employee is a public overloading operator member function that overload the plus (+) operator. This member function would have definition to add the data member of two object of the Employee class together. For example, if signer and co-signer are both employees, then their monthly salary would be added together and stored in the third object which represent the combined monthly salary. The combined monthly salary would later be used to calculate to the chance of the approval of the mortgage.

3. SelfEmployed Class

SelfEmployed class is a subclass of the superclass Worker. The SelfEmployed class would inherit from the Worker class and add another data member taxable_annual_salary. It is because the different salary payment, so that self-employed worker is different from employee. Self-employed worker generally don’t have consistent salary every month as employee does, and self-employed work is usually not as stable as employee, because they usually receive salary as contract based, therefore the mortgage company would use annual taxable income as the basis to determine self-employed workers. Therefore, in this program, I differentiate employee and self-employed workers based on how salary type. The taxable_annual_salary would later to convert to monthly salary to calculate with other data to determine the chance of approval of the mortgage. The SelfEmployed class would have a constructor to store the data members and additional member function for setting and getting the taxable_annual_salary data member. SelfEmployed class would have another different definition for the get_description() member function so that the program can use polymorphism to determine which specific one to use.

- Data Member:
taxable_annual_salary is a private double data member of the SelfEmployed class. taxable_annual_salary would be used to store the taxable annual salary of self-employed worker, and it would then be converted to monthly salary for calculation in order to determine the chance of the approval of the mortgage. The console would ask the user if he/she is employee or self-employed. If the user choose self-employed, the object of SelfEmployed would be called and the value of the annual taxable salary would be stored in the object of this class.

- Member Functions:

The constructor SelfEmployed(name_ : string, house_price_ : double, property_tax_ : double, monthly_debt_ : double, down_payment_ : double, credit_score_ : int, taxable_annual_salary_ : double) is public and it would be used to store the values of the data members. The data members would then be stored in the object of the SelfEmployed class and be used to calculate the chance of the mortgage approval as well as generate a txt report in the end.

set_taxable_annual_salary(taxable_annual_salary : double) is a public void member function that would be used to set the value of the taxable_annual_salary based on the input parameter that would get from the console after asking the user to input the taxable annual salary amount.

get_taxable_annual_salary() is a public double member function would output the value of the taxable annual salary stored in an object of the SelfEmployed class. It would later be used for calculation to determine the chance of the approval of the mortgage.

get_description() is a public string member function that has the same name and type as the member function in the Worker superclass and the Employee subclass. The member function has a different definition other than the one in the Worker class and the one in the Employee class. The definition would include such as worker’s name, house price, property tax, monthly debt, down payment and taxable annual salary of the self-employed worker. The program would use polymorphism to determine which specific definition of the function to call.

combine(employee : Employee&, selfemployed : SelfEmployed&) is a public friend void member function that uses the data members from both the Employee class and the SelfEmployed class. It is second declaration in the SelfEmployed class, and it would has a definition in a separate implementation file as mentioned earlier. The function is used when the signer and co-signer are different types of worker. Therefore this is used to convert annual salary of the self-employed worker to monthly salary and add together to employee’s monthly salary for the calculation of the chance of the approval.

Whole Program Description:

This program is belong to extra credit EC1 category that this program has practical usefulness that it can compute useful mortgage information that can be used in real world by many people who intend to buy a house.
This Housing Mortgage Analyzer would have three header files for the three classes and three implementation files for these classes. There will be also one more implementation file for the definition of the friend function. In the main.cpp, I would ask the user for name, house price, property tax, monthly debt and down payment. I would then ask if he/she is employee or self-employed; I would also include descriptions of their meanings. I would then ask the user to input either monthly salary or taxable annual salary based on their input. I would call the corresponding class object constructor and store the values to its data members. Then I would use these data members to calculate the chance of approval of a conventional mortgage.

Initially the chance would be 100%, if the monthly income-to-debt ratio is above 50%, then the chance would be -100%, therefore the total chance of approval would be zero. If the monthly income-to-debt ratio is lower than or equal to 50% but higher than 43%, then the chance of approval would be -30%. If the monthly income-to-debt ratio is below or equal to 43% and above 36%, then the chance of approval would be -10%. And if the income-to-debt ratio is equal to or below 36%, there will be no deduction to the initial 100% chance.

Secondly, the console would determine the deduction percentage based on worker’s credit score, its deduction percentage listed below:

<table>
<thead>
<tr>
<th>Credit Score</th>
<th>Deduction Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal or above 750</td>
<td>0%</td>
</tr>
<tr>
<td>Equal or above 650 but below 750</td>
<td>20%</td>
</tr>
<tr>
<td>Equal or above 550 but below 650</td>
<td>50%</td>
</tr>
<tr>
<td>Below 550</td>
<td>70%</td>
</tr>
</tbody>
</table>

If there is cosigner, then the approval percentage would be average among all signers of the mortgage.

The program would validate user’s input and output a final report to write in a new txt file which include all the information about the purchaser and co-signers if there is any, and the chance of the approval of a conventional mortgage.

The program chance of approval is for estimate only, the final result has to be determined by the official bank and financial institutions.

Encapsulation (chapter 14)- Worker.h (Line 16 – 50), Employee.h (Line 16 – 27), SelfEmployed.h (Line 15 – 27)
Inheritance (chapter 15)- Employee.h (Line 15, 19-21), Self-Employed.h (Line 14, 18-21)
Polymorphism (chapter 15)- Worker.h (Line 42 – 50), Employee.h (Line 25), Employee.cpp (Line 16 – 25), SelfEmployed.h (Line 25), SelfEmployed.cpp (Line 16 – 25), main.cpp (Line 375, 378, 383, 387)
Static data members and functions (chapter 16)- Worker.h (Line 16, 39-41), Worker.cpp (Line 65), Employee.h (Line 21), SelfEmployed.h (Line 20)
Friend functions (chapter 16)- Employee.h (Line 26), SelfEmployed.h (Line 27), friend.cpp (whole file), main.cpp (Line 262, 299)
Overload operators (chapter 16)- Employee.h (Line 27), Employee.cpp (Line 28-32), SelfEmployed.h (Line 26), SelfEmployed.cpp (Line 28 -32), main.cpp (Line 233, 328)
Flex Project Description

This program allows users to look up individual player stats, look up team stats, display league leaders, compare stats of 2 players, compare stats of 2 teams, and display the statistical leaders of a team.

This program reads from 2 .txt files, one of which has every player in the league and their per game stats, the other containing every team and their per game stats.

A player object is created for every player and added in to a vector.

A team object is created for every team and added in to a vector.

**Quick descriptions of user options:**

- **Search by Player**
  Searches for player (by first and last name) in a vector containing all players
  Prints that player’s statistics to the console

- **Search by team**
  Searches for team (by team name) in a vector containing all teams
  Prints that team’s per game statistics to the console

- **Display League Leaders**
  Sorts vector of players using sort algorithm and friend compare functions (there are 5 different compare functions for each of the 5 statistics)
  Prints the top 10 league leaders in all 5 statistical categories

- **Team Leaders**
  Searches for players that are members of that team in the vector of players
  A Team object is created and filled by members of that team.
  Prints that team’s leaders in all 5 statistical categories
- Compare the statistics of two players

Searches for 2 players (by their first and last names) in the vector of players

Prints (to the console) the statistics of two players found and puts “**” around the higher number denoting which player had better numbers for that statistic. If stats are equal no “**” are used.

- Compare statistics of two teams

Searches for 2 teams (by team name) in the vector of teams

Prints (to the console) the statistics of two teams found and puts “**” around the higher number denoting which team had better numbers for that statistic. If stats are equal no “**” are used.

**Encapsulation:**

Data members and member functions from the team, player, and athlete classes would demonstrate encapsulation.

Encapsulation is demonstrated in:

Player.h; lines 17-64
Athlete.h, lines 17-40
Team.h, lines 18-65

**Inheritance:**

Player class would inherit from the athlete class (first name, last name, position, height). Will use object composition between Team and Player classes.

Inheritance is demonstrated in:

Player.h ; line 15

Object composition is demonstrated in:

Team.h; line 20
Polymorphism:

A function, get_info(), that will be called for the Athlete class will display first name, last name, and position.

When the function is called for the Player class it will display first name, last name, city, team, height, and statistics.

Polymorphic virtual get_info() function is declared in:

    Athlete.h; line 38
    Player.h; line 45

Defined in:

    Athlete.cpp; lines 52 - 60
    Player.cpp; lines 112 – 126

Used in:

    Main.cpp, line 1024

Static data members/functions:

Static int player_count keeps track of the total amount of players in the league

Static int player_count is declared in:

    Athlete.h, line 32

Defined in:

    Athlete.cpp, line 13

Used in:

    Athlete.cpp, line 24
    Main.cpp, line 918
**Friend functions:**

Friend functions were used as “compare functions” for the sort algorithm. I used 5 different friend functions, for each of the 5 statistics. The sort algorithm was then used to sort the vector of players by points, rebounds, assists, blocks, or steals.

Friend compare functions were declared in:

```
Player.h; lines 58 -62 and in main.cpp; lines 21 – 25
```

Defined in:

```
Main.cpp lines 1183 – 1211
```

Used in:

```
Main.cpp lines 427, 444, 459, 474, 490
```

**Overload operators:**

Used overloaded == operator to compare players and teams

Player overloaded == operator declared in:

```
Player.h; line 54
```

Defined in:

```
Player.cpp; lines 130 – 237
```

Used in:

```
Main.cpp; line 909
```
Team overloaded == operator declared in:

    Team.h; line 63

Defined in:

    Team.cpp; lines 297 – 405

Used in:

    Main.cpp; line 1152

For error checking I customized the console.cpp file that was used earlier in the semester.

**Extra Credit:**

I would like to attempt EC1 (if you believe it is applicable)
Because of space formatting issues with Word I did not include the arrows indicating inheritance or object composition between the classes

<table>
<thead>
<tr>
<th>Athlete</th>
</tr>
</thead>
<tbody>
<tr>
<td>-First Name</td>
</tr>
<tr>
<td>-Last Name</td>
</tr>
<tr>
<td>-Height</td>
</tr>
<tr>
<td>-Position</td>
</tr>
<tr>
<td>-static player_count keeps track of the number of players in the league</td>
</tr>
<tr>
<td>+string get_first_name()</td>
</tr>
<tr>
<td>+string get_last_name()</td>
</tr>
<tr>
<td>+string get_height()</td>
</tr>
<tr>
<td>+string get_height()</td>
</tr>
<tr>
<td>+string get_position</td>
</tr>
<tr>
<td>+set first_name(string)</td>
</tr>
<tr>
<td>+set last_name(string)</td>
</tr>
<tr>
<td>+set height(string)</td>
</tr>
<tr>
<td>+set position(string)</td>
</tr>
<tr>
<td>+virtual void get_info()</td>
</tr>
<tr>
<td>display Athlete’s first name, last name, and position</td>
</tr>
<tr>
<td>+Athlete(string, string, string, string)</td>
</tr>
</tbody>
</table>
**Player class inherits from Athlete Class**

<table>
<thead>
<tr>
<th>Player</th>
</tr>
</thead>
<tbody>
<tr>
<td>-points per game</td>
</tr>
<tr>
<td>-assists per game</td>
</tr>
<tr>
<td>-rebounds per game</td>
</tr>
<tr>
<td>-blocks per game</td>
</tr>
<tr>
<td>-steals per game</td>
</tr>
<tr>
<td>+double get_points()</td>
</tr>
<tr>
<td>+double get_assists()</td>
</tr>
<tr>
<td>+double get_rebounds()</td>
</tr>
<tr>
<td>+double get_blocks()</td>
</tr>
<tr>
<td>+double get_steals()</td>
</tr>
<tr>
<td>+string get_team()</td>
</tr>
<tr>
<td>+void set_team()</td>
</tr>
<tr>
<td>+void set_points(double)</td>
</tr>
<tr>
<td>+void set_assists(double)</td>
</tr>
<tr>
<td>+void set_rebounds(double)</td>
</tr>
<tr>
<td>+void set_blocks(double)</td>
</tr>
<tr>
<td>+void set_steals(double)</td>
</tr>
<tr>
<td>+void get_info() will display Player’s first name, last name, height, position, and statistics</td>
</tr>
<tr>
<td>+ overloaded == operator will compare the stats of two players and indicate which player had higher numbers</td>
</tr>
<tr>
<td>+friend bool compare_points(player1, player2)</td>
</tr>
<tr>
<td>+friend bool compare_rebounds(player1, player2)</td>
</tr>
<tr>
<td>+friend bool compare_assists(player1, player2)</td>
</tr>
<tr>
<td>+friend bool compare_blocks(player1, player2)</td>
</tr>
<tr>
<td>+friend bool compare_steals(player1, player2)</td>
</tr>
<tr>
<td>+Player(string, string, string ,string , string, double, double, double, double, double, double, double)</td>
</tr>
</tbody>
</table>
Team class uses object composition with Player Class

<table>
<thead>
<tr>
<th>Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Team name</td>
</tr>
<tr>
<td>-City</td>
</tr>
<tr>
<td>-Roster a vector containing the players on a team</td>
</tr>
<tr>
<td>+set_team_name(string)</td>
</tr>
<tr>
<td>+set_city(string)</td>
</tr>
<tr>
<td>+get_team_name()</td>
</tr>
<tr>
<td>+void add(player) adds a player to the roster (vector) data member</td>
</tr>
<tr>
<td>+string get_roster() returns a string listing all the players on the team</td>
</tr>
<tr>
<td>+string get_city()</td>
</tr>
<tr>
<td>+Player points_leader() returns the team leader (a Player) in points</td>
</tr>
<tr>
<td>+Player rebounds_leader() returns the team leader (a Player) in rebounds</td>
</tr>
<tr>
<td>+Player blocks_leader() returns the team leader (a Player) in blocks</td>
</tr>
<tr>
<td>+Player assists_leader() returns the team leader (a Player) in assists</td>
</tr>
<tr>
<td>+Player steals_leader() returns the team leader (a Player) in steals</td>
</tr>
<tr>
<td>+overloaded == operator returns the Team that averages the most points</td>
</tr>
<tr>
<td>+Team(string, string, double, double, double, double, double, double)</td>
</tr>
<tr>
<td>+Team()</td>
</tr>
</tbody>
</table>
Edits are in red.

Console:

You are using my photography assistant V1.0

Welcome!

What kind of Camera user are you?
   1. Full frame user.
   2. APSC user.
   3. Micro Four Thirds user.

Enter the number related to your camera:

Does your camera have in-body stabilization? (Please enter 'y' or 'n'.)

What kind of Lens are you going to be using? (Focal Length)

Does your Lens have stabilization? (Please enter 'y' or 'n'.)

What is the largest aperture of your lens? (Lowest F-number)

Thank you! What tool would you like to use?

   1. DoF Calculator.
   2. Ideal Shutter Speed Calculator.

//Polymorphism will take place after user decides on what kind of Calculator they would like to use.
---------------------------------------------------------------------------------------------------------------------
This is the DoF Calculator.

Thank you for your input, I just need to ask you one more question, what would be your distance from the subject in meters?

//After the input I should get:

Negative values for far limit and total depth of field means your camera and lens combo can focus to infinity, which is great for landscapes, but not very desirable for portraits.

//I also reworded the bottom four outputs, so it feels more friendly to the user, rather than just a calculator.
Hyperfocal distance:
Near limit of acceptable focus:
Far limit of acceptable focus:
Total depth of field:

// I thought pasting the following lines into this document would make better sense, although I already had this idea in the UML diagram, I just did not display it here. I also renamed getNoOfCalculatorObjects to getNoOfCalculatorRuns just because I thought it would make more sense for someone reading the code.

cout << "You have used my tool " << Calculator::getNoOfCalculatorRuns() << " time(s) since running it. " << endl << endl;

cout << "Would you like to use the photography assistant V1.0 again? (Please enter 'y' or 'n'). ";

This is the Ideal Shutter Speed Calculator.

Your Ideal Shutter Speed for your Camera and Lens combo is....
Your Camera and Lens have this amount of stabilization stops:

// I thought pasting the following lines into this document would make better sense, although I already had this idea in the UML diagram, I just did not display it here. I also renamed getNoOfCalculatorObjects to getNoOfCalculatorRuns just because I thought it would make more sense for someone reading the code.

cout << "You have used my tool " << Calculator::getNoOfCalculatorRuns() << " time(s) since running it. " << endl << endl;

cout << "Would you like to use the photography assistant V1.0 again? (Please enter 'y' or 'n'). ";

Calculations for DoF Calculator:

Hyperfocal distance:
\[
H = \frac{f^2}{b} + f
\]

Near distance of acceptable sharpness:
\[
D_n = \frac{s(H - f)}{H + s - 2f}
\]

Far distance of acceptable sharpness:
\[
D_f = \frac{s(H - f)}{H - s}
\]
Depth of field = Far limit - Near limit.

\[ H \] is the hyperfocal distance, mm

\[ f \] is the lens focal length, mm

\[ s \] is the focus distance

\[ D_n \] is the near distance for acceptable sharpness

\[ D_f \] is the far distance for acceptable sharpness

\[ N \] is the f-number (related to aperture)

\[ c \] is the circle of confusion, mm

We can assign circle of confusion based on sensor size.
Full Frame: 0.03m or 30mm.
APSC: 0.02m or 20mm.
Micro Four Thirds: 0.015m or 15mm.

Calculations for Ideal Shutter Speed Calculator:

Ideal shutter speed = \((1 / \text{focal length}) \times (1 \text{ or } 1.5 \text{ or } 2 \text{ based on sensor size}) \times (\text{multiplied by } 1 \text{ or } 2 \text{ based on the availability of Lens stabilisation}) \times (\text{multiplied by } 1 \text{ or } 2 \text{ based on the availability of Lens stabilisation})\).

I have planned out this project and it should hit the following concepts:

- Encapsulation (chapter 14).
- Inheritance (chapter 15). *(Polymorphism depends on Inheritance).*
- Polymorphism (chapter 15). *(Explained above and in UML diagram).*
- Static data members and functions (chapter 16). *(I have used Static data members and functions to count the number of calculators running).*
- Friend functions (chapter 16). *(I used a friend function to access two private variables from the member class and this will allow me to calculate the amount of stabilization stops).*
- Overload operators (chapter 16). *(Overloading operator to write to file).*

It should also hit these requirements:
- At least two classes, and no more than four. This means you may have two to four header files and their accompanying interface files.
- Each class can have at most five (5) data members and ten (10) member functions. This includes static data members and functions.
● You must have at least one subclass which inherits from one of the other classes. Use public, protected, and private access to data members and functions as needed. (My ShutterSpeedCalculator class inherits from my Calculator class).
● Your program must read from or write to text files, and interact with the user through the console. (I will write the outputs to a text file, this could later be printed for practical purposes).
● You need to ensure that valid input is provided using exceptions and error checking.

It should also hit the extra credit for being a practical program or educational program since it could be described as either.

Here is my UML diagram for the project (Added members and functions are in red):

References:

http://www.dofmaster.com/equations.html
Overview:
My Flex Project will be to create a stat entry system for the Tampa Bay Lightning Hockey Club.
EC1: “Create a program with practical usefulness.”

My program will allow you to add players to a roster, edit their stats, and print out the stats upon the user’s request entered through the command menu. The program will do calculations for adding the stats to the current stats read from a file as well as calculations for the goalies save percentage and goals against average.

This program in its current state is a demonstration build and is only a few minor adjustments away from being able to be used with multiple files based on the team requested, so it could be used to store every team in the NHL when asked which team you wanted to enter stats for, or any hockey team you wanted to track stats for from any league. The only reason I have it set up for just the Tampa Bay Lightning is to make it easier to test for grading. But the potential that this program has, I believe, qualifies it as a practical usefulness program for EC1.

UML Diagram:
**Class Descriptions:**

**Team:**
The Team class is the head class for this program. It stores the vectors for the skaters and goalies, stores the team record, add players, removes players, and prints different types of data representation.

- **int wins**
  - number of wins

- **int losses**
  - number of losses

- **int overtime_losses**
  - number of overtime losses

- **static int num_players**
  - number of players on the team

- **vector<Skater> skaters**
  - stores objects of type Skater

- **vector<Goalie> goalies**
  - stores objects of type Goalie

- **static int get_num_players()**
  - gets num_players

- **void add_player()**
  - this function is used to add players to the system. If the player is already in the system, the player is not added and a message appears

- **void remove_player()**
  - removes a player that the user specifies

- **void print_team_stats()**
  - prints a list of players and their stats in a chart

- **void top_5_goals()**
prints the top 5 skaters leading the team in goals

- void top_5_assists()
  - prints the top 5 skaters leading the team in assists

- void top_5_points()
  - prints the top 5 skaters leading the team in points

- void add_game()
  - This is the function used primarily after all the players are entered.
  - This function will ask the user the result of the game, update the record, and go through each player for their stats to be entered for that game.

+ friend void read_file(Team&)
  - This function will be a global function that is friended with the Team class in order to get access to the private vectors.
  - When the program starts read_file() will be called so that the data in the file is stored in the private vectors.

+ friend void write_file(Team&)
  - This function will be a global function that is friended with the Team class in order to get access to the private vectors.
  - When the user specifies to exit the program, write_file() is called to take the information from the private vectors and write it to a file.

+ void run()
  - This function wraps everything together and guides the user using a command menu.
    - This is what could be in main() but I decided to make it a function instead.

**Player:**

The Player class is the parent class for Skater and Goalie. This class holds all the information that applies to any player entered.

# string first_name
  - stores the first name

# string last_name
  - stores the last name

# int number
stores the jersey number

# int games_played
stores the number of games the player has played

+ Player(string, string, int)
  Player constructor

+ void set_first_name(string)
  first_name setter function

+ void set_last_name(string)
  last_name setter function

+ void set_number(int)
  number setter function

+ void set_games_played(int)
  games_played setter function

+ string get_first_name()
  first_name getter function

+ string get_last_name()
  last_name getter function

+ int get_number()
  number getter function

+ int get_games_played()
  games_played getter function

+ virtual void print_player()
  overwritten by Skater or Goalie class upon call

+ bool operator== (Player&)
  This operator overload will check to see if two players are the same by comparing the first_name, last_name, and the number.
This will be used in the add_player() and remove_player() functions to check if the player trying to be added or removed is already in the system.

**Skater:**

The Skater class is the subclass of the Player class. It holds all the Skater specific information.

- int goals
  - stores number of goals

- int assists
  - stores number of assists

+ Skater(string, string, int, int, int, int)
  - Skater constructor

+ void set_goals(int)
  - goals setter function

+ void set_assists(int)
  - assists setter function

+ int get_goals()
  - goals getter function

+ int get_assists()
  - assists getter function

+ int points()
  - calculates and returns number of points

+ void print_player()
  - Overwrites the function from the Player class and prints out the information in the Skater format.

**Goalie:**

The Goalie class is the subclass of the Player class. It holds all the Goalie specific information.

- int minutes
  - stores the minutes played
- int seconds
  o stores the seconds played

- int shots_against
  o stores the number of shots against

- int saves
  o stores the number of saves made

+ Goalie(string, string, int, int, int, int, int, int, int, int)
+ void set_minutes(int)
  o minutes setter function
+ void set_seconds(int)
  o seconds setter function
+ void set_shots_against(int)
  o shots_against setter function
+ void set_saves(int)
  o saves setter function
+ int get_minutes()
  o minutes getter function
+ int get_seconds()
  o seconds getter function
+ int get_shots_against()
  o shots_against getter function
+ int get_saves()
  o saves getter function
+ double goals_against_average()
  o calculates the goals against average and returns it
+ double save_percentage()
- Calculates the save percentage and returns it

```c++
+ void print_player()
```
- Overwrites the function from the Player class and prints out the information in the Goalie format.

**Techniques Used:**

**Encapsulation:**
The data members for Player, Skater, and Goalie will either be protected or private. The vectors in Team will be private.

Team.h: 20-37
Player.h: 11-16
Skater.h: 8-11
Goalie.h: 8-13

**Inheritance:**
Player is the parent class of the Skater and Goalie classes. Every player has a name, number, and games played, but the position specific stats will be in their respective subclass.

Skater.h: 6
Goalie.h: 6

**Polymorphism:**
`void print_player()` will be a virtual function in the Player class and will be overwritten by the same function in the Skater and Goalie classes. I did this because the different types of Player have different information to print out with the regular player information.

Player.h: 33
Skater.h: 25
Skater.cpp: 34-40
Goalie.h: 32
Goalie.cpp: 63-72
Team.cpp: 447, 457
Static Data Members and Functions:

There will be static data member num_players that counts the number of players added, and static function int get_num_players() will get the number of players.

Team.h: 13, 29
Team.cpp: 3-6, 183, 317, 409, 419, 988, 1005

Friend Functions:

friend void read_file() and friend void write_file() will be global functions friended to the Team class so it has access to the private vectors that they need to complete their tasks.

Team.h: 44-45
Team.cpp: 927-1103
main.cpp: 15, 19

Overload Operators:

I will be overloading the == operator and using it in the add_player() and remove_player() functions to determine if the player being added or removed is already in the system by comparing name and number.

Player.h: 35
Player.cpp: 52-66
Team.cpp: 63, 76, 406, 416
Hello. I understand that changes here were rather drastic compared to what was laid out with Submission 1, but I was under a time crunch when I didn’t expect this program to be as complicated as it would be if I didn’t make some changes. I did not expect to heavily rely on interactions between the Enemy and Avatar class, which lead to massive changes in parameters. The Avatar class needed more functions to shorten itself, even though it was already lengthy. While the Avatar class may have too many functions, it is pretty offset by the Boss and Enemies class having little to no functions now due to Enemy being the abstract class. Thank you for your understanding!

- Added main file to document and functionality
- Changes made to Avatar Class
  - Clarified how many values were used in stats data member (7). Speed stat removed from vector<int> stats. Considered unnecessary complications.
  - equipment changed to vector<pair<string, int>>.
  - items changed to int<pair<string, int>>.
  - private int temp_defense added.
  - 1 constructor added: Avatar(string = Xenon).
  - Added get_name() function.
  - Added get_level() function.
  - Added action(Enemy&, int &) function.
  - attack() function has Enemy& parameter.
  - questions() function has Enemy& parameter. Only manipulates enemy stats now. No analyze functionally due to analyze() function being removed.
  - flee() function now takes in integer.
  - die() function returns an int instead of being void.
  - level_up() function now has an Enemy parameter. Mention how current EXP stat is changed in level_up function.
  - Added equip() function.
  - Added print_items() function. Also writes out to “Inventory.txt” file.
  - Added drop_item() function.
  - Added pick_up() function.
- Added valid_choice function.
  - fight function now takes in two reference parameters of Avatar and Enemy.

- Changes made to Enemy Class
  - Considered an abstract class now because random_choice() is now a pure virtual function.
  - Moved here EXP static int data member from Enemies.
  - Added get_name() function.
  - Moved here get_item() function from Enemies.
  - Moved here get_EXP() function from Enemies.
  - defend() function removed. Considered not important in Enemy AI.
  - analyze() function removed. Considered not an important advantage for the user.

- Changes made to Enemies Class
  - Removed all data members.
  - Created a constructor that takes in an int.
  - Has overriding random_choice function.
  - die() function removed. Considered unnecessary as the function is already in the Enemy Class.

- Changes made to Boss Class
  - Created a constructor
  - Added overriding random_choice function.
  - die() function removed. Considered unnecessary as the function is already in the Enemy Class.
  - fight() function removed. Function was edited in Enemy.cpp to work for both Boss and Enemies objects.

**Important Requirements**

- Encapsulation
  - Where: Enemy.h
  - Line #: 21 – 24

- Inheritance
  - Where: Enemies.h
  - Line #: 6
  - Where: Boss.h
  - Line #: 8

- Polymorphism
  - Where: Enemy.h
  - Line #: 30
  - Where: Enemies.h
  - Line #: 10
  - Where: Enemies.cpp
  - Line #: 80 - 99
Summary
You are a character who is playing through an entire RPG dungeon. This entire RPG adventure is all text-based (it’s up to the user to fill in the gaps with their imagination). This dungeon is crawling with various enemies and one final boss. However, you will gain EXP, collect items, and use OOP and trivia knowledge to help you along the way. The game will end when you die or you defeat the final boss.

Outline
1 Main + 4 Classes

- Main
  - Read in “Intro.txt”
  - Tell user about default name. Ask if they want another name, which will lead to creating the one Avatar() object.
  - Explain by reading in “Tutorial.txt”
  - Allow user to choose between using the Menu or Exploring. This is repeated until user dies or is level 10.
  - If the user reaches level 10, then a Boss() obj will be created. The user can either die from the fight, or win and read in the “End_Credits.txt” file.
  - read_file(const string&) function reads in any text file regularly.
- valid_choice(char, char, char) function checks to see whether an entered character matches either of two required characters.

- **Enemy**
  - Superclass of Boss and Enemies
  - **4 Data Members (Protected)**
    - Name (string): Contains the enemy name.
    - Stats (vector<int> (6)): Contains the current HP, max HP, critical HP, Attack stat, Defense stat, and Grip stat.
    - Item (pair<string, int>): An item that the enemy will have and can use.
    - **EXP (static int):** EXP amount the Enemy can drop.
  - **9 Member Functions**
    - Get Name: Returns enemy’s name.
    - Get Drop: Gives the drop data member to the user. Occurs when enemy dies and has a drop.
    - Get EXP (static int): Transfers set EXP to user. This function is static as it’s returning a static data member.
    - Random Choice (Pure Virtual).
    - Attack: Uses a normal attack.
    - Use Item: Only turn-based move that isn’t random and uses the enemy’s healing item.
    - Die: **Returns an int to see whether current HP <= 0.**
    - Operator[]: Allows easier accessing of values in the Stats vector via subscripting.
    - Fight (friend definition): Allows a fight to occur between an Avatar and an Enemies objects. Also allows a fight between an Avatar and a Boss objects.

- **Boss**
  - Subclass of Enemy
  - Will spawn once the user reaches level 10.
  - **Updated Data Members of Enemy Superclass**
    - Name (static string): Ragnarok
    - Item (pair<string, int>): Will 100% be a Med-kit.
  - **1 Constructor + 2 Member Functions**
    - **Boss()**: Initializes data members and seeds random number generator.
    - Random Choice: Calls the roll dice function or use item function if current HP <= critical HP.
    - Roll Dice: Certain events will occur depending upon the value of the dice (2-12).
      - 2 (Snake Eyes) – Reduces user HP to 1.
• 3 (Attack) – Normal Attack.
• 4 (Swap) – Swaps Avatar and Boss’ Stats. Attack and Defense swapped only.
• 5 (Unexpected) – Boss damages himself.
• 6 (x1.03) – Give the boss a slight boost in this Attack and Defense stat (x1.03).
• 7 – (Laser Beam) – Performs a normal attack with 2x the damage.
• 8 (Triple Attack) – Performs a normal attack 3 times. (Doubles temporary defense for user to half damage)
• 9 (*Snap*) – Halves the user’
• 10 (Attack) – Normal Attack.
• 11 (HP Steal) – Steal some HP from the Avatar. (20% of current HP)
• 12 (Restoration) – Heals half of the Boss’ HP.

• Enemies
  o Subclass of Enemy
  o 1 Constructor + 2 Member Functions
    ▪ Constructor: Takes in avatar level parameter. Initializes values by reading in enemies from “Enemies.txt” file. Also seeds in random number generator.
    ▪ Random Choice: Chooses which function to perform during an enemy’s turn. Will 100% choose Use Item if current HP <= critical HP.
    ▪ rps: Chooses a random Avatar stat to affect. Plays rock-paper-scissors with the user. Will continue until a tie doesn’t happen. Who ever wins gets a stat increase or decrease in favor.

• Avatar
  o 6 Data Members
    ▪ Name (string): Contains avatar name. Default name is Xenon. Can choose name at the start of the game.
    ▪ Stats (vector <int>(7)): Contains the current HP, max HP, current EXP, EXP cap, attack stat, and defense stat, and evasion stat.
    ▪ Level (int): Contains the avatar level. Default level is 1. Max level is 10.
    ▪ temp_defense (int): Will serve as a temporary defense stat. Only functionally used in defend() function.
    ▪ Equipment (vector<pair<string, int>> (3)): Contains the weapon, helmet, and body armor. These will increase the attack and defense stats respectively.
- Items (vector<pair<string, int>> (5)): Items you can use while in exploring-menu or in-battle. Limit of 5 items in inventory at a time not including equipped equipment.

  - 1 Constructor + 18 Member Functions
    - Constructor: Uses 1 string as a parameter that initializes name. Initialize other data members. Seeds in random number generator.
    - Get Name: Returns name.
    - Get Level: Returns level.
    - Action: Allows a user to choose from a set of battle actions: attack, questions, use Item, defend, or flee.
      - Attack (option): Use a normal attack.
      - Questions (option): You are first asked what stat you want to manipulate(always enemy). You are asked a question about OOP or just questions about anything (4 options). If successful, the desired stat decrease will occur. If unsuccessful, the opposite of the desired stat decrease will occur.
      - Use Item (option): Use a healing item from inventory.
      - Defend (option): Increases the defense stat for one turn. Uses the temp_defense data member.
      - Flee (option): Try to escape current fight. Succession depends upon evasion stat and grip stat of the enemy. 0% success with the boss.
      - Menu (exploring-option): Allows you to view stats, equipment, and items. Furthermore, you can change equipment. Moreover, you can drop items or heal. Will be given as user reaches level 10, which is the reason for using the reference parameters to Avatar and Boss.
      - Explore (exploring-option): Chance of finding and possibly collecting item. If inventory is full, user will be allowed to replace an item in inventory. There is a chance to random encounter with basic enemy. Nothing can occur.
      - Die: Returns a int to determine if the game ends. Returns 1 when current HP <= 0. Returns 0 otherwise.
      - Level Up: Occurs always after each fight win, actual leveling up occurs when current EXP + get EXP(enemy) >= EXP cap. Increase max HP, EXP cap, attack stat, defense stat, speed stat, and evasion stat. Sets current HP to max HP. Set current EXP to (current EXP + get EXP(enemy)) % EXP cap.
      - Fight (friend): Occurs randomly when using the explore option. It’s a friend function so that every enemy can easily receive the user’s stats without using parameters. Also, enemies engage fights, not the user. Defined in Enemy.cpp file.
- Equip: Allows user to switch out equipment (or equip when having no equipment). Equipment will have a visible effect in fights and when viewing stats in menu.
- Print Items: Prints out a list of items in inventory. Furthermore, this list is written out to the “Inventory.txt” file.
- Drop Item: Drops an item in inventory.
- Pick Up: Allows a user to pick up an item from exploring or an enemy drop.
- Valid Choice: checks to see whether an entered character matches either of two required characters.

Items will be read in from text-files

- Weapons
  - Boxing Gloves (+10 Attack)
  - Diamond Pickaxe (+20 Attack)
  - Shotgun (+50 Attack)
  - Drill (+80 Attack)
  - Time Harp (+100 Attack)
  - Holy Lance (+130 Attack)
  - Bane of Light (+160 Attack)
  - Lightning Hammer (+200 Attack)
  - Lightsaber (+250 Attack)
- Armor
  - Red Cap (+10 Defense)
  - Overalls (+20 Defense)
  - Metal Helmet (+20 Defense)
  - Knight’s Armor (+50 Defense)
  - Iron Suit (+160 Defense)
  - Mask of Terrible Fate (+100 Defense)
  - Yellow Jumpsuit (+200 Defense)
- Healing
  - Sunrise Tomato (+50 HP)
  - Bandages (+100 HP)
  - Elixir (+150 HP)
  - Med-Kit (+200 HP)

Enemies (will also be read-in from a text file) (Stats in-file)

- Filthy Peasant
- Noble Raccoon
- Walking Tree
- Whimsical Owl
- Mighty Shovel
• Ultra Ghost
• Purple Guy
• Elite Penguin
• Energetic Cat-Girl

End Credits

• Will be written to a text file and console.
### Avatar

- name: string
- stats: vector<int>(7)
- temp_defense: int
- level: int
- equipment: vector<pair<string, int>>(3)
- items: vector<pair<string, int>>(5)

+ Avatar(n : string)
+ get_name() : string
+ get_level() : int
+ action(e : Enemy&, flee_check : int&) : void
+ attack(e : Enemy&) : void
+ questions(e : Enemy&) : void
+ use_item() : void
+ flee(grip : int) : void
+ defend() : void
+ menu(a : Avatar&, r : Boss&) : void
+ explore(a : Avatar&) : void
+ die() : int
+ level_up(e : Enemy&) : void
+ equip(equip_choice : string) : void
+ print_items() : int
+ drop_item() : void
+ pick_up(item : string, item_stat : int) : void
+ valid_choice(check_char : char, chr1 : char, chr2 : char) : int
+ fight(a : Avatar&, e : Enemy&) : friend void
### Enemy

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td>string</td>
</tr>
<tr>
<td><code>stats</code></td>
<td>vector&lt;int&gt;({6})</td>
</tr>
<tr>
<td><code>item</code></td>
<td>pair&lt;string, int&gt;</td>
</tr>
<tr>
<td><code>EXP</code></td>
<td>static int</td>
</tr>
<tr>
<td><code>getName()</code></td>
<td>string</td>
</tr>
<tr>
<td><code>getDrop()</code></td>
<td>pair&lt;string, int&gt;</td>
</tr>
<tr>
<td><code>getEXP()</code></td>
<td>static int</td>
</tr>
<tr>
<td><code>attack(a_HP : int&amp;, temp_defense : int, a_name : string)</code>: void</td>
<td></td>
</tr>
<tr>
<td><code>useItem()</code></td>
<td>void</td>
</tr>
<tr>
<td><code>die()</code></td>
<td>int</td>
</tr>
<tr>
<td><code>operator[](index: int): int&amp;</code></td>
<td></td>
</tr>
<tr>
<td><code>fight(a : Avatar&amp;, e : Enemy&amp;)</code>: void</td>
<td></td>
</tr>
</tbody>
</table>

### Enemies

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Enemies(avatar_lvl : int)</code></td>
<td>N/A</td>
</tr>
<tr>
<td><code>random_choice(a_stats : vector&lt;int&gt;&amp;, temp_defense : int&amp;, a_name : string)</code>: void</td>
<td></td>
</tr>
<tr>
<td><code>rps(a_stats : vector&lt;int&gt;&amp;, temp_defense : int&amp;, a_name : string)</code>: void</td>
<td></td>
</tr>
</tbody>
</table>

### Boss

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Boss()</code></td>
<td>N/A</td>
</tr>
<tr>
<td><code>random_choice(a_stats : vector&lt;int&gt;&amp;, temp_defense : int&amp;, a_name : string)</code>: void</td>
<td></td>
</tr>
<tr>
<td><code>roll_dice(a_stats : vector&lt;int&gt;&amp;, temp_defense : int&amp;, a_name : string)</code>: void</td>
<td></td>
</tr>
</tbody>
</table>
Flex project submission 1

EC2

Turn based combat RPG

User selects a combat partner from 3 options. Computers partner is randomly selected for each battle.

User then navigates through gameboard by choosing different routes to the end, throughout each path the user will have to battle the computer at various stops while also being able to pickup and use different items to help them in battle.

Classes: Partner, Player, Combat

1. Partner:

String Choice
Vector<String> Moves
Int Health – health of combat partner
Int level -level of combat partner
Int exp – level up counter

Functions:
Partner();
string get_choice();
vector<string> get_moves();
void set_choice();
void set_moves();
void set_health();
void set_boss_health();
int get_health();
int get_exp();

2. Player:

Partner cp
String Items

Functions:
string get_partner();
vector<string> get_items();
void pick_partner();
void set_partner(Partner);
void pickup_item(string);
void Use_item(int); - updates item list
friend void check_level_up(Partner&); - checks the exp to see if the partner is ready for leveling up
vector<string> operator+=(string i) – adds an item to the list
int get_health() returns the health
vector<string> get_moves() returns the moves
void operator-(int i) subtracts health
void set_health() – sets the partners health
Partner& get_cp() return cp
int get_level() return cp.get_level()
Player& operator+(int i) adds to cp exp
void increase_exp(int) – increases exp

*most combat functions will be static so that they can be called easily in the main function without having to declare a combat object. This will improve the readability and organization of the program by allowing me to organize most of the code in the static battle functions. This will make the main function clean and easily readable because most of the code will be in the static functions and will just need to be called when necessary. The static functions will accept a player object and use it for the battles*

3.Combat: will inherit from player so that it can have direct access to modify the data members of a player object. This will cut down on some blocky code by allowing me to modify data members without having to only have the functionality of the coded member functions of the player class.

Vector<vector<char>> Gameboard
int Px, Py; - holds the players location

Functions:
void get_position(); - returns the position
void set_board(); - reads the gameboard from a text file and stores it in an array
void print_board(); - prints the board
static bool Battle(Player&); - most of the code is in here and controls the battle
void Move_forward(); - moves the player on the board
void reset(Combat); - resets the player at the beginning
Static void Use_item(Player&); - updates the item list
bool check_end(Player&); - checks if the game is over
void check_battle(Player&); - checks if a battle should be started
void check_item(Player&) – checks for an item on the board
static bool Boss(Player&) – code for the boss fight
**Player**

- Partner cp;
- Items;

- string get_partner();
- vector<string> get_items();
- void pick_partner();
- void set_partner(string);
- void use_item(string);
- friend void check_level_up(Partner);
- vector<string> operator++(string);
- int get_health();
- vector<string> get_moves();
- void operator=(int i);
- void set_health();
- Partner& get_cop();
- int get_level();
- void operator=(int i);
- void increase_exp;

**Partner**

- string choice;
- vector<string> moves;
- int health;
- int level;
- int exp;

**Combat**

- vector<vector<char>>* gameboard;
- int Px, Py;

- void get_position();
- void set_board();
- void print_board();
- static bool battle(Player&);
- bool check_end(player&);
- bool check_battle(player&);
- void check_item(player&);
- static bool boss(player&);
Flex Project

The project is a simple representation of a University. As an example, we can assume it to be “University of South Florida”.

A University is composed of various people. The people can be of various types. For this project, we will assume that 2 types of people are there in University.

1. Student
2. Instructor

Each person has a name, gender and age. The person class will be polymorphic and will behave as Student or Instructor object appropriately when called.

A Student class will have a GPA while an instructor will have a rating.

When a student do work, they would generally be completing their homework.

When an instructor do work, they would generally be grading the students papers.

So we will show the polymorphic behavior of the do_work() function from the Person class.

A University also contains various departments. A department has a name, size, address and list of students enrolled and list of professor who teach in this department.

Classes to be designed

Department.cpp
Department.h

Defines the Department class. Exact structure will be shown in UML diagram.

Fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Stores the name of the Department</td>
</tr>
<tr>
<td>size</td>
<td>Stores the area of the Department</td>
</tr>
<tr>
<td>address</td>
<td>Stores the address of the Department</td>
</tr>
</tbody>
</table>

Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getName()</td>
<td>Returns the name of the Department</td>
</tr>
<tr>
<td>setName()</td>
<td>Sets the name of the Department</td>
</tr>
<tr>
<td>getSize()</td>
<td>Returns the area of the Department</td>
</tr>
<tr>
<td>setSize()</td>
<td>Sets the area of the Department</td>
</tr>
<tr>
<td>getAddress()</td>
<td>Returns the address of the Department</td>
</tr>
<tr>
<td>setAddress()</td>
<td>Sets the address of the Department</td>
</tr>
<tr>
<td>overload &lt;&lt;</td>
<td>Friend function</td>
</tr>
</tbody>
</table>
Person.cpp
Person.h

Defines a person class. Exact structure will be shown in UML diagram.

**Fields:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Stores the name of the Person.</td>
</tr>
<tr>
<td>age</td>
<td>Stores the age of the Person.</td>
</tr>
<tr>
<td>gender</td>
<td>Stores the gender of the Person.</td>
</tr>
</tbody>
</table>

**Functions:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getName()</td>
<td>Returns the name of the Department</td>
</tr>
<tr>
<td>setName()</td>
<td>Sets the name of the Department</td>
</tr>
<tr>
<td>getAge ()</td>
<td>Returns the area of the Department</td>
</tr>
<tr>
<td>setAge ()</td>
<td>Sets the area of the Department</td>
</tr>
<tr>
<td>getGender ()</td>
<td>Returns the address of the Department</td>
</tr>
<tr>
<td>setGender ()</td>
<td>Sets the address of the Department</td>
</tr>
<tr>
<td>print()</td>
<td>Prints the person to Console. This will be a virtual function that can be overridden in the derived class.</td>
</tr>
<tr>
<td>do_work()</td>
<td>Do the work on the person object. This will be a virtual function that can be overridden in the derived class.</td>
</tr>
</tbody>
</table>
| overload operator to print | Friend function  
                  | Prints the Person to the console.                                            |
|              | This is same as print() function but can be cascaded in cout stream, so a different usage. |
| normalizeName() | This will be friend function that will normalize the name of the University object. Capitalize the first character in every word. |
Defines the Instructor class. This class will be inherited from the Person class as public inheritance. This class will over-ride the do_work and print function appropriately.

**Additional Field:**

| rating   | Stores the rating of the Instructor |

**Functions:**

| getRating () | Returns the rating of the Instructor |
| setRating () | Sets the rating of the Instructor |
| print()     | Prints the Instructor to Console. The definition is overridden in the Instructor class. |
| do_work()   | Do the work on the Instructor object. The definition is overridden in the Instructor class. |
| overload << operator to print | Friend function Prints the Instructor to the console. |
|             | This is same as print() function but can be cascaded in cout stream, so a different usage. |

**Student.cpp**  
**Student.h**

Defines the Student class. This class will be inherited from the Person class.

**Additional Field:**

| gpa       | Stores the gpa of the Student |

**Functions:**

| getGPA () | Returns the gpa of the Student |
| setGPA () | Sets the gpa of the Student |
| print()   | Prints the Student to Console. The definition is overridden in the Student class. |
| do_work() | Do the work on the Student object. The definition is overridden in the Student class. |
| overload << operator to print | Friend function Prints the Student to the console. |
This is same as print() function but can be cascaded in cout stream, so a different usage.

**University.cpp**

**University.h**

Defines the University. A university will contain a list of persons and a list of department.

**Fields:**

<table>
<thead>
<tr>
<th>name</th>
<th>Stores the name of the University.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector&lt;Person*&gt;* persons</td>
<td>Stores the vector of person in University. This will contain all Students as well as Instructor.</td>
</tr>
<tr>
<td>Vector&lt;Department*&gt;* departments</td>
<td>Stores the vector of Department in University. This will contain all departments</td>
</tr>
</tbody>
</table>

**Functions:**

<table>
<thead>
<tr>
<th>printDepartments ()</th>
<th>Prints all department info on Console</th>
</tr>
</thead>
<tbody>
<tr>
<td>printPeople ()</td>
<td>Prints all people info on Console</td>
</tr>
<tr>
<td>addPerson()</td>
<td>Add person to University</td>
</tr>
<tr>
<td>overload += to add person</td>
<td>Overloaded operator</td>
</tr>
<tr>
<td>addDepartment</td>
<td>Add Department to University</td>
</tr>
<tr>
<td>overload += to add department</td>
<td>Overloaded operator</td>
</tr>
</tbody>
</table>

**Util.cpp**

**Functions:** Utility functions for various operations. These functions are collected together for use anywhere throughout the program. We can use them anywhere we want and in any class without the need to create an object of the Util class. We just need to include the header file and the function would be available. Example: To generate something randomly (like student age, or rating, or gpa) will have to include Util.hpp as a header class and the function can be used.

<table>
<thead>
<tr>
<th>getRandom(n)</th>
<th>Generates a random number between 1-n</th>
</tr>
</thead>
<tbody>
<tr>
<td>getValidInt(prompt,min)</td>
<td>Takes input from user for a int value</td>
</tr>
</tbody>
</table>
max) between min and max. If user would enter invalid value, it will prompt to enter correct one. This won't let the program crash.

The prompt is the message shown to the user.

getValidInt(prompt) Takes input from user for a int value. If user would enter invalid value, it will prompt to enter correct one. This won't let the program crash.

The prompt is the message shown to the user.

**Menu.cpp**

Provides function to provide menu option to user.

Static function display that can be used to display the menu to user. Every time, we need to display menu, we don’t need to create a menu object but use the static method of display which is static with syntax as Menu::display()

**main.cpp**

Driver class for the project that will demonstrate the whole program.
UML Diagram

The UML diagram of this project is explained below. I have used a freeware to prepare this diagram.
Demo of line numbers where C++ feature is demonstrated.

- Encapsulation (chapter 14)
  - Each class we have private/public/protected members.
  - Person.hpp – Has protected members. (line 12)
  - Student.hpp – Has private members. (line 12)
  - University.hpp – Has private members. (line 17)
  - Department.hpp – Has private members (line 12)

- Inheritance (chapter 15)
  - Student.hpp - Student class shows public inheritance from Person. (Line 10)
  - Instructor.hpp - Instructor class shows public inheritance from Person. (Line 10)

- Polymorphism (chapter 15)
  - Main.cpp
    - normalizeName Line 27, 32, 42, 47
    - overloaded << shows polymorphic behavior Line 28, 33, 43, 48

- Static data members and functions (chapter 16)
  - Menu.hpp
    - Line 11

- Friend functions (chapter 16)
  - Person.hpp
    - normalizeName function – Line 35
    - check implementation in cpp file where private data is accessed.
  - Department.hpp
    - normalizeName function – Line 34
    - check implementation in cpp file where private data is accessed.

- Overload operators (chapter 16)
  - University.hpp
    - += for person (Line 37) – Check cpp class for implementation detail.
    - += for department (Line 39) – Check cpp class for implementation detail.