Program: Watopoly
Group Members: Alan Zhou, Richard Zhu, Taiqian Zhang

Introduction:
Watopoly is a board game of real estate transactions at the University of Waterloo. Players try their best maintaining their asset advantage and force other players into bankruptcy through rolling dice and purchasing / trading buildings. The winner is the only one who does not go bankrupt(dropping off) in the end.

Overview:
We control the execution and user input of the program through main.cc, and we display the game board after each command through the TextDisplay class. The Watopoly class serves as a controller of the whole game and wraps up all the classes that have relationships with it. The Player class creates and keeps track of the player objects, and the Building class helps represent each grid on the game board, and it has two subclasses - "Property" and "NonProperty". The Property class helps represent each grid that is ownable, including the Residence, the Academic(Building), the Gym, where each of them is the subclass of Property. The NonProperty class help represent each grid that is not ownable, including the GooseNesting, the CoopFee, the SLC, the NeedlesHall, the Tuition, the (Collect)OSAP, the DCTL (DC Tims Line), the GoToTims, where each of them is the subclass of NonProperty. Finally, the Dice class creates the dice object and determines the player's rolling numbers in each round.

Main.cc:
The main.cc file controls the execution flow of our program and handles user input. It is the platform where clients interact with the program. Necessary variables and objects are initialized here.

Watopoly class:
This class serves as a "controller" of the game who wraps up the objects and methods from other classes. In main.cc, we only operate on an object created from the Watopoly Class called "game", since other classes should not be exposed to the client due to the purpose of encapsulation.

Player class:
This class creates player objects for our game. It is responsible for keeping track of individual player attributes and methods are written to support player actions. Some methods are accessors to help return values from private fields to reinforce encapsulation.

Building class:

This is an abstract class representing each grid on the game board. It has two subclasses “property” and “non-property”. We have many virtual functions in this class waiting to be overridden by its subclasses according to their own unique features. For example, the action method here is virtual since different individual building objects will perform different actions.

Property class:
This class inherits from the Building class and represents each grid that is purchasable, including the Residence, the Academic(Building), and the Gym, which are its 3 concrete subclasses. It provides some methods that can be shared commonly by these subclasses, and some virtual methods that can be overridden. For instance, the get_price() method in the class is virtual because the charging rules for the Residence, the Academic(Building), and the Gym are different.

NonProperty class:
This class inherits from the Building class and represents a grid that cannot be purchased by players, as opposed to the Property class. Each NonProperty object performs its unique actions, therefore the action() method here is virtual and is waiting to be overridden by its concrete subclasses such as CoopFee, SLC, and NeedlesHall, etc.

Dice class:
This class creates a dice object and decides the rolling numbers in each player’s turn. We referenced the idea from shuffle.cc to generate the possibilities that can be used in the rolling process.

TextDisplay class:
This class creates a TextDisplay which is similar to what we made in a4q3. It is responsible for printing out the current game board. It gets notified every time when the display needs to be changed. For example, when an improvement is built, or when a player moves.

Updated UML:
https://app.diagrams.net/#G1CBailNeeBpyxpD9oCudDyQ8N8k2PftO1

Design:
Our design of the program strictly follows the 5 most important concepts about OOP programming, which are Encapsulation, Inheritance, Polymorphism, Abstraction, and Composition. Moreover, in the process of designing our code, we put an emphasis on achieving low coupling and high cohesion.

Encapsulation:

In each of our classes, we try to avoid exposing our fields to the client by making them private, or protected if necessary to grant access to subclasses. Meanwhile, we provide public getter methods to access these private attributes in a read-only manner. Finally, we encapsulate methods from different classes into methods written in our Watopoly(controller) class which is in charge of the gameplay process. Thus, in main.cc the Game object created from the Watopoly class is the only object we need to interact with. This way, the program successfully allows for better control over access to the data and hiding implementation details from the outside world.

Inheritance:
The advantage of Inheritance is the ability of a class to inherit properties and methods from a parent class. Child classes can inherit, override, or extend the functionality of the parent class. For example, Academic Buildings and Residences are derived classes from the Property class, which itself is a derived class of the abstract Building class. With each level of inheritance, we add more detailed and unique features to our objects while maintaining some common patterns that can be applied to all Building objects. This significantly reduces code duplication.

Polymorphism:
We achieved polymorphism by writing a number of virtual methods in base classes, who are overridden in subclasses. This allows for similar but different objects to respond to the same method call in their own ways. For example, the Action() method is a virtual method written in the abstract class Non-Property, which owns a list of derived classes such as SLC, Needles Hall, and Collect Osap that override this method in their own rules to perform individual actions accordingly. Polymorphism makes code reusable and easier to maintain.

Abstraction:
The Watopoly game is a complex system. To break it down into smaller and more manageable parts, we create abstract base classes as well as concrete subclasses to represent a hierarchical organization of our objects. This makes our program more intuitive for both the people writing the code and reading the code since it reflects logic from the real world. For example, Building is an abstract object, Gym object can be a building and so can a SLC object. However, we never create an actual object from our abstract class, but its concrete subclasses instead. We use a Building pointer to indicate its essential type of being a “Building”.
Composition:
In contrast to inheritance, composition promotes a “own a” relationship instead of “is a”. We can use composition when we want to merge objects from different classes that do not have a hierarchical relationship together to accomplish some tasks. For example, our game controller class Watopoly always composes a dice object, 2-7 player objects, and 40 building objects, which are essential elements to create a game. That is, when a Watopoly object is created, the objects that it is composed of are also created. Accordingly, when the Watopoly object is destroyed, the objects that it is composed of are destroyed. Composition provides great flexibility

to our program, we can easily add extra features to the game by adding more composed objects for the Watopoly class.

Some challenges we encountered and how we solved them:

1. Memory Leak:
   We encountered numerous memory leaks when using regular pointers in the first place due to the length of the program(hard to keep track of all the heap allocated memory). We solved the problem by implementing shared pointers since it is convenient for its own memory management. With smart pointers, the idea of wrapping every resource in a stack allocated object also follows the principle of RAII.
2. Deep nested loops:
   At first we had a number of very deep nested loops in our main function, which makes our code harder to read and debug(lots of cases where infinite loops happen). We simplified it by encapsulating some tasks into smaller methods to make the structure of our code more organized and readable. Also, for a few nested loops that are not really dependent on each other, we managed to flatten them into single loops, which reduces the chances of bugs.
3. Large method that handle too many situations:
   In some classes, we firstly wrote some methods that handle too many conditions, which eventually turns out to be very difficult to read and debug. To solve this, we break them down into smaller methods that handle a more specific case. The original "big" method becomes a general method that invokes these "small" methods.
4. The order of what to code first:
   At first, we did not plan the order of writing what first very clearly. This causes many conflicts and slows down our progress. For example, with Alan writing Player class, Taiqian writing Watopoly class, and Richard writing the main function at the same time, we do not have a chance to look at our teammate's code before implementing our own idea. This leads to a lot of problems when merging code, and a lot of work has to go in vain. Therefore, we spent some time discussing the order of implementation(such as writing the main function first before all other classes), and the rest of the process has become much smoother.
5. Passing by copies:
   There has been an occurrence where we want to change what a pointer points to in a method. However, we failed to do so and encountered segmentation fault in the beginning since our getter method did not return a reference, therefore we were passing in a copy of the pointer that we wanted to modify to our function, and this bug took a long time to find. We fixed it by returning a reference to the pointer in our accessor.

Resilience to Change:
Our program is adaptive to changes because of its low coupling and high cohesion. When writing our classes, we pay extra attention to SRP(single responsibility principle). Most of our classes focus on its own unique responsibility and should not highly depend on other classes. For example, our dice class is only in charge of the rolling process, that is, the Dice object only comes into use when a player needs to roll, but nothing else. Moreover, the good use of encapsulation prevents different classes easily modifying each other's fields. By avoiding adding friend classes and making as many fields private as possible and providing public getter methods, we enhance low coupling by hiding the internal details of a class and exposing only a clean, stable interface to other classes(low class dependencies). Furthermore, we make good use of composition in our game controller, Watopoly class(we favor composition over inheritance when we only need a "owns a" relationship, since it brings less coupling). Therefore, whenever we want to add extra objects to the game, we can just simply write another class module and add it to our Watopoly class.

Answers to Questions:
Question 1:
After reading the subsection, implementing the Observer Design Pattern is the first thing that comes to mind. Since it is intuitive to set textdisplay and Building as observers subscribing to Building like what we did in a4q3 (each cell has its neighbors as observers and a textdisplay observer). So that every time a building changes its state(e.g. an improvement is built), its observers react accordingly. However, after thorough discussion among group members, we found out that implementing the whole Observer Design Pattern was not necessary, because this time each building block is independent of the others, therefore we do not need to attach other building objects as observers. So it is not necessary to separately create a subject and observer class because textdisplay is the only object in the program that needs to be notified. Thus, we still draw lessons from the Observer Design pattern to implement a notify method in the Textdisplay class, whenever we call the move(), improve(), or remove_player() methods in the Watopoly class and the states that have an impact on Textdisplay are changed, we notify Textdisplay to change its printing result accordingly.

Question 2:
Suppose we want to model SLC and Needles Hall more closely to Chance and Community Chest cards. A suitable design pattern we can use is the Template Design Pattern. This way, we can create a general super class called "card" that have some default features for the subclasses "SLC" and "Needle Hall", while in the meantime leaving specific behaviors to be implemented individually by its concrete subclasses. For example, our "card" superclass can be used to handle the operation of drawing or shuffling cards, and our "SLC" and "Needle Hall" cards class will be in charge of performing unique actions for each card. However, according to the actual

requirement of the project, we did not implement the Template pattern as it is not necessary. Instead, we created a vector of integers and referenced the idea from shuffle.cc to generate the instructed probabilities for SLC and Needle Hall behaviors. For example, in a vector of {1,1,2,2,2,3,3,3,3,3}, after shuffling, we collect the value at v[0], getting a 1 represents the probabilities of 1/5, a 2 being 3/10, and a 3 being 1/2.

Question 3:
Technically a decorator pattern could be fitted because its purpose is to add or modify features of an object. In our case, the academic building is the object and improvements are features to be added on. However, we believe implementing the Decorator Design Pattern is not necessary. Unlike the pizza example discussed in class, the improvements here are simple and consistent across all buildings without requiring their own unique set of behaviors. Therefore, we decided using a vector of integers storing the building's charging price at each improvement level is the more convenient way. We can determine the current charging price by accessing the index of the vector depending on our improvement_level field. In our actual final implementation, we chose to use an array instead of a vector since we know that the cost for each improvement and the total amount of available improvement is fixed. In comparison, an array is more efficient than a vector(an array is a simpler data structure with a fixed size).

Extra Credit Features:

1. No memory leak without explicitly managing memory:
   In our program, we handled all memory management via vectors and smart pointers. There are no "delete" statements.
2. Additional commands for better game play:
   In our program, we provide the user an option of the "help" command, which gives the user the hint of all his/her available commands in case the user is not familiar with the game. There is also a "quit" command that enables the user to exit the game.
3. Enhanced board display:
   When a property is mortgaged, there will be a sign "M" appearing on the upper right of the block.

Final Questions:
1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs:

- The most important thing I learned from this group project is that communication between team members is key. It is crucial to have a planning session in the very beginning with everyone attending to ensure each person understands their roles and responsibilities. Frequent meetups are also necessary and helpful so that everyone gets to share their progress or any challenges they are facing, this helps a struggling group member get help in time. Furthermore, from this experience our group got to learn how to collaborate on github and git, these could be very useful tools for our future career. There would definitely be plenty of projects that we need to work in a group after we enter the workplace, so getting relevant experience at school is certainly helpful. What's more, agreeing to a general coding standard or habit could also be important, otherwise trying to read and understand your teammate's code may be a problem. Although It is not very likely for every group member to follow the exact same standard, it is still vital to have a discussion about this before starting, trying to get everyone feeling comfortable. Last but not least, writing comments is very critical, it saves a lot of time for both you and your teammates when reviewing code or debugging.

- If working alone, the major thing I learned is how to generally approach a large program like this. First of all, it is important to first read through the entire thing carefully to make sure that I understand all the requirements of the program. Secondly, I must plan before starting, it is easy to find yourself in a dead end halfway through if you do not plan ahead. Thirdly, try to break down large tasks into smaller ones and solve them one by one, this makes the entire working process easier and more focused. Also, writing comments while writing code so that I can easily keep track of the old work I have done to avoid unnecessary confusion. Last but not least, come with sufficient test cases to test all possible scenarios, which could get tedious but it is very necessary.

2. What would you have done differently if you had the chance to start over:

- We should have learned how to collaborate on git and github sooner. In the very beginning, we were exchanging our updated code on messages, which was very inconvenient and error-prone.
- We should have met up more often. We realized the amount of time we met up was not enough, if the group members could meet regularly, conflicts could be resolved more quickly and necessary help could be provided in time. It is often the scenario that each group member spent a lot of time completing tons of work, but then identifying numerous issues when merging, and lots of changes need to be made. Connecting more frequently would definitely make our work more efficient.
- We should have gone through the questions more carefully in the very beginning. We started without completely understanding every requirement, which led to us having to make big unnecessary changes to our code halfway through.
- We should have started sooner. For larger programs, debugging usually takes longer. We did not start early enough to reserve time for code reviewing and debugging, which made our time really tight in the end. We had to pull all-nighters to finish debugging everything.

Conclusion:
This project has been a challenging but fun and meaningful experience for every three of us. This is the first time for all of us to build a whole game from scratch in a small group. It is a good exercise for not only our coding ability and understanding about OOP, but also developing teamwork skills. The Watopoly game is a well-designed project that carries out the content we learned from lectures into real life software development. We made use of topics discussed in class like encapsulation, polymorphism, design patterns, and smart pointers, etc, which are all fundamental concepts in the field of programming.