

**《High-level Language Programming Project》Report**

Project Name: Monopoly

**School :** Computer Science and Engineering

**Major :** Computer Science and Technology Full English Creative Class

& Computer Science and Technology

**Student Name :** 谢东欣、黄嘉威、赵旻昆

**Teacher :** Xuemiao Xu

**Submission Date:** 2021.6.30

|  |  |
| --- | --- |
| Teacher comment | signature：  date： |
| Performance evolution |  |
| remark |  |

**Background**

Monopoly is a multiplayer strategy board game which had the aim of exposing the capitalism’s harm originally and simulates the management for process nowadays. The participants who gain equal money at the beginning need to make good use of their fortune and transaction strategies to purchase lands and build buildings in order to acquire rents from other players. In addition, there will be only one winner at last while others all go bankrupt, as a consequence of which the name of this game is Monopoly.

**Motivation**

1. It is an appropriate choice to enhance our capability of problem analysis and programming skill since realizing it requires rigorous logical thinking and huge amounts of work.
2. We can also have a better understanding of object-oriented programming through doing this project due to the fact that it contains a variety of classes and inheritance relations etc.
3. Most people are familiar with this game as well as being fond of it, so that our project can draw people’s attention and gain their approval easily.
4. The topic also creates an opportunity for us to improve our innovative thinking ability. Because we can enrich our game’s functions and improve the practicality by standing on the players’ side since we already know the classical rules of the game.
5. There is no doubt that we students who major in computer science are always under great study pressure which does harm to our health and games like Monopoly can reduce our stress and bring happiness to us.

**Game rules design**

**(1)The beginning of game：**

The map contains 22 cities and 4 players.

The origin of the players is Guangzhou.

Each player has an asset of 10000 yuan originally.

**(2)The Duration of Sound：**

We use the duration as the alternative of rolling the dice which contains six kinds of points to decide the number of each step randomly. In this case, players need to press the record button and make some voice each time.

**(3)Property：**

Once a player comes to an area which has no owner, he/she can decide whether to buy it or not. And once a player comes to an area which has already had an owner, he/she has to pay the corresponding taxes.

**(4)Houses：**

Players can choose to build some buildings in the cities that he/she owns as long as he/she has enough money. In this case, other players that come to this city need to pay more taxes.

**(5)Cash withdrawal：**

Players can sell a house at a lower price than it was built. Also, players can sell a city at a lower price than it was bought.

**(6)Transaction：**

Players can make a transaction with their opponents on the area where has no building at any time.

**(7)Bankrupt：**

A player will go bankrupt when his/her money become negative even if he/she sell or mortgage all his/her property.

**(8)Success condition：**

The only player who has not gone bankrupt at last wins.

**Runtime Environment**

**Windows 10, Qt 5.9**

**Duty Assignment**

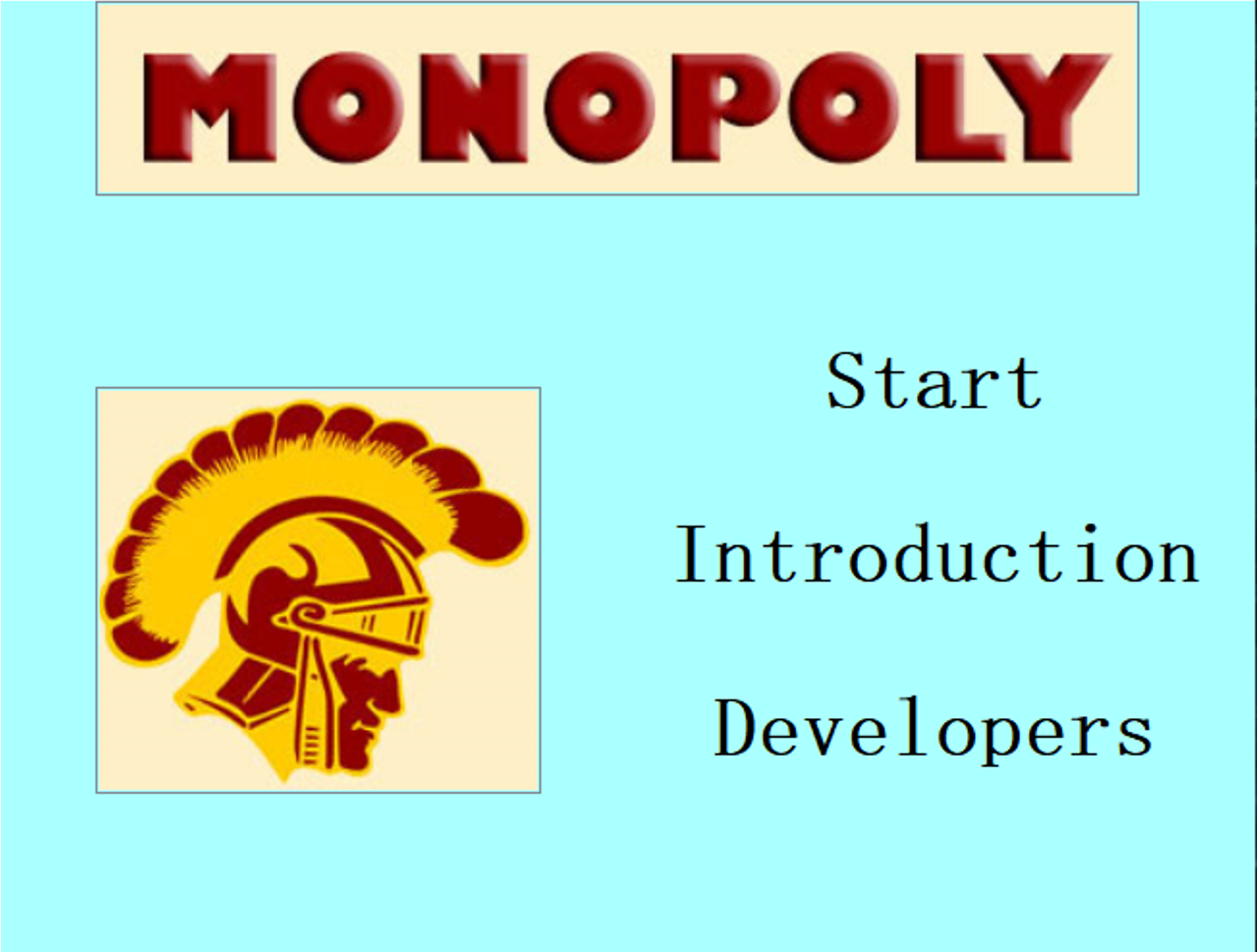
Huang Jiawei took charge of the kernel part. He wrote the source codes of class Player, class City, class Unit, class Map and class Control. For each class, he edited the data members and functions to realize the necessary results that we expected.

Xie Dongxin designed the game window and connected the modules in the game window with the game kernel. He not only made humanized design of the game window, but also realized quite a few difficult functions such as controlling the players to move in the map, showing the prompt information and the corresponding events triggered by pressing the buttons etc.

Zhao Minkun realized the recording function that the voice dice needs, designed the windows apart from the game window and perfected the visualization of the game window. As for the perfecting part, specifically speaking, he added the labels on the units and realized the corresponding functions. In addition, he realized the function of restart the game and added a little animation etc.

**Qt UI design**

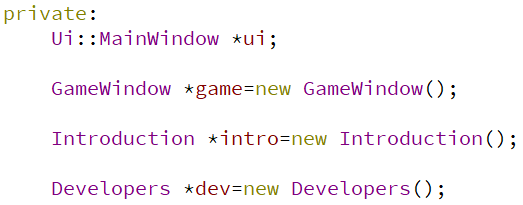
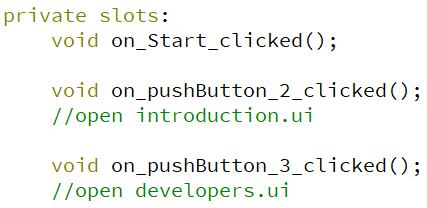
1. **mainwindow.ui**



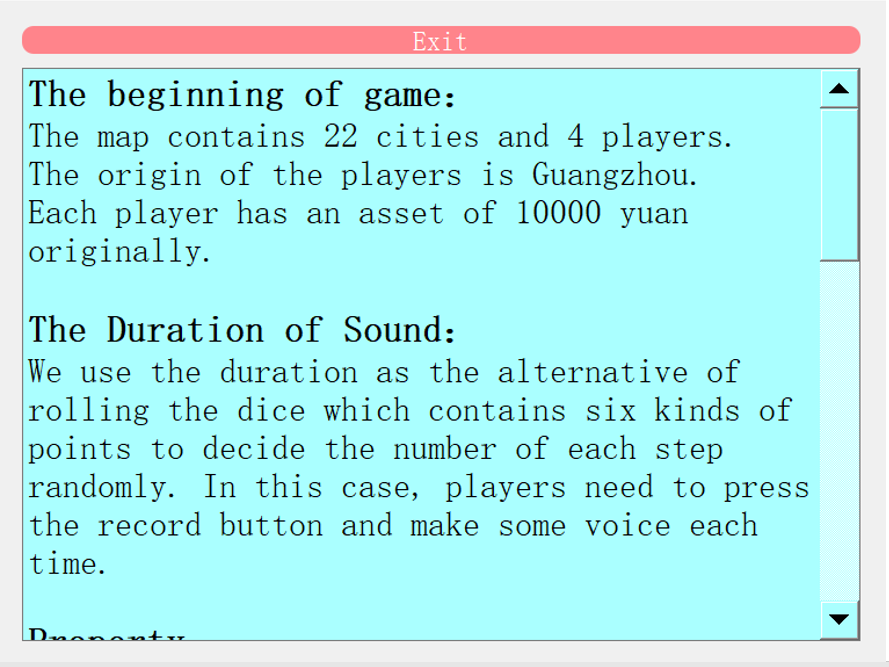
This is the first window shown when our project is opened.

The title on the top and the picture on the left are QGraphsViews modules and the menu on the right contains three buttons. Here is a technique that lets the buttons merge into the background better. That is ticking the flat option in the property setting part.

By pressing these three buttons, this main window will be closed and the corresponding window will be opened. In order to realize it, we defined these slots functions.



1. **introduction.ui**



This window will be shown if the Introduction button in the main window is clicked. Also, you can also return back to the main window by pressing the exit button on the top.

As is shown above, this introduction window is for conveying the game rules and playing method to the players.

1. **developers.ui**



This window will be shown if the Developers button in the main window is clicked. Also, you can also return back to the main window by pressing the exit button on the top.

As is shown above, this developers window contains we three group members’ pictures.

1. **gamewindow.ui**

First of all, we design the architecture of QT. First of all, we use QT designer to design gui. We firstly try to use the layout provided by QT designer, but it is hard to control the size of pushbutton we used to symbolize the chess lattice. So finally we use the freedom layout without official layout, The chess lattice consist of the scenery of city as background, Top left corner is the head portrait of the owner of city and the top right corner is the number of buildings on city which represented by a number. The chess lattice circle on the top left corner of the whole main window. The top right corner of the whole main windows is the head portrait which now is 4 players. The part below these head portrait is the amount of players and the part below this part is the bulletin board which presents the information of the game procedure. the bottom left part of main window is the ability push button of game such as buy, build, sell and update.



**About the programing part of ui**

First:

We use the code below to connect push button with the city by “for loop”, so we can avoid connect 22 chess lattice with city in list by 22 lines code.

chess Ps = { ui -> pc1 , ui -> pc2 , ui -> pc3 , ui -> pc4 } ; // this is the list of player push button.

Q List < Q Push Button \*> buttons = this -> find Children < Q Push Button \* > () ; // this is the all push button list

for (int i = 0 ; i < 22 ; i++ )

{

connect ( buttons[i] , &Q Push Button :: clicked , this , [=]

{

this -> target \_ city \_ show (i) ;

});//this function connect city with push button.

}

ui -> target \_ text -> set Text (Q String :: from Std String ( to \_ string (con -> get X() -> ta)));// and we use this code line to show information on the bulletin board

Others types of push button we use groove function specifically call the ability we need.

private slots:

void **on \_ pushButton \_ 28 \_ clicked** ();

// The button "Next"

void **on \_ Buy \_ local \_ clicked** ();

void **on \_ Dice \_ clicked** ();

void **on \_ sell \_ place \_ clicked** ();

void **on \_ sell \_ house \_ clicked** ();

void **on \_ Build \_ clicked** ();

void **on \_ exit \_ clicked** ();

**the connection with game kernel**

About the connection with game kernel, First of all, we design the most important push button “ Next ”.

if( ! haDice )

{

ui -> target \_ text->set Text ("You have never dice!");

return;

}

else

{

ha Dice = false;

if ( ! con -> round())

{

ui -> target \_ text -> set Text ("You can't finish your round!");

}

}

At first we check If this player has dice and move, if not, he can’t continue to the next player, he need to dice at first. When after check and pass, the fee will be checked if it is enough to pay if the city he attend has owner and the owner is not himself. if enough, the balance on amount will decrease the fee. without enough money, the now player will fail to pass to next player

int temp = ( clock () – this -> start Time);

if ( temp < 1000 ) { temp %= 100 ; }

else { temp /= 1000 ; }

this -> start Time = 0 ;

players [ now \_ player]-> location move(temp , X -> get length ());

ind = players [ now \_ player] -> get location ();

return temp;

Secondly, we design the dice push button with two kinds. On the one hand, the number of dice is depend on the last time you dice, between 0 s and 1 s, the number of dice will be randomly chosen between 0 and 100. when the last time large than 1 s, the number of dice will be the floor of last time.

If you click each city, the information of this city will be shown in the information text browser, include City Name, the Number of buildings, the Owner of this city, the Cost of city if you want to buy this city, the Fee if player stay on the place which is not belong to him, the Fee will become if the next building was built and the cost of building a hotel. The concatenation of string is achieved by the code below in Map.cpp

We also add an update line at the end of each function: “update \_ see \_ money ();”. And the content of function is:

“

string t1="";

for (int i=0;i<4;i++)

{

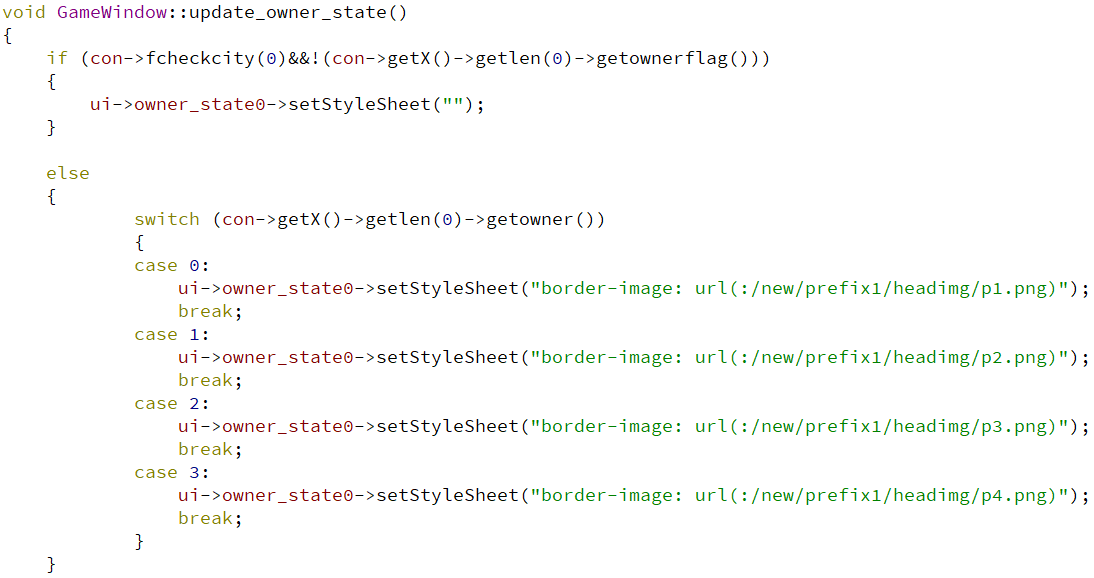
t1 += "p" + to \_ string (i)+": "+to \_ string (con -> players [i] -> get amount ()) + "\n";

}

ui -> see \_ money -> set Text (Q String :: from Std String(t1));

“

**the methods of displaying the cities’ state of owner and number of building in the cities**



Here, we defined a function called update\_owner\_state and it will be invoked when the players buy cities or sell cities. And it repeats the procedure shown above for 22 times in order to update the state of owner of all the cities.

In each procedure, we all invoked the setStyleSheet function to update the state of owner. In addition, we set the corresponding player’s picture to the label on the top left corner if the city does have an owner and set the label nothing otherwise.



Here, we defined a function called update\_building\_number and it will be invoked when the players build houses or sell houses. And it repeats the procedure shown above for 22 times in order to update the number of buildings of all the cities.

In each procedure, we all invoked the setText function to update the number of buildings.

1. **dicebysound.ui**



This window will by shown when the player clicks the Dice button.

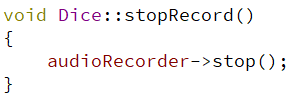
As the rules have introduced, we use the duration of sound as the alternative of the points of a dice, which is a significant innovation point of our project.

In this window, the system will begin to record when the upper button is pressed and the system will stop recording when the lower button is pressed.

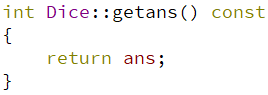


The method of recording is shown above and it is in the constructor of Dice class.

Not only did we use the QAudioRecorder to realize recording sound, but we also invoked QAudioProbe. Later we connected them together in order to realize the detection function without which we cannot gain the physical quantity we need. What’s more, after setting the path, format and name of the audio file, we invoked the record function in the last line to record sound.

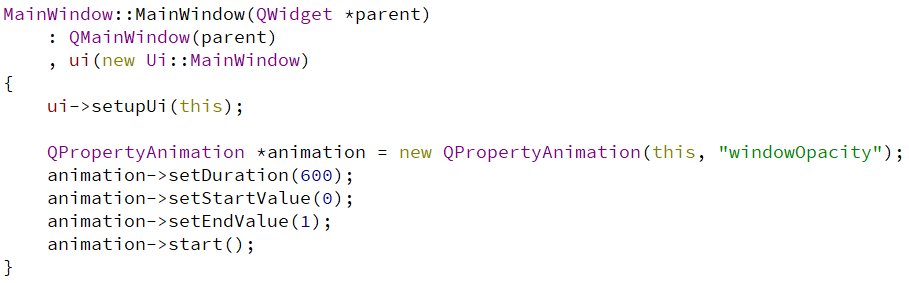


Invoke the stop function to stop recording when the Stop button is pressed.



Here, we made the duration of sound to be the seed in order to realize the randomness and fairness of dice. Addtionally, it is worth mentioning that we defined a static variable called ans concerning the difficulty of passing the value among so many classes.

1. **Animation of windows’ entry**



In order to improve players’ sense of experience, we added a little animation of windows’ entry in the constructors of UIs as is shown above.

The essence of this animation is just setting the opacity of the windows from 0 to 100% gradually. In this case, the result is that the windows appear in the screen with desalination animation.

**The basic logic and kernel of Monopoly**

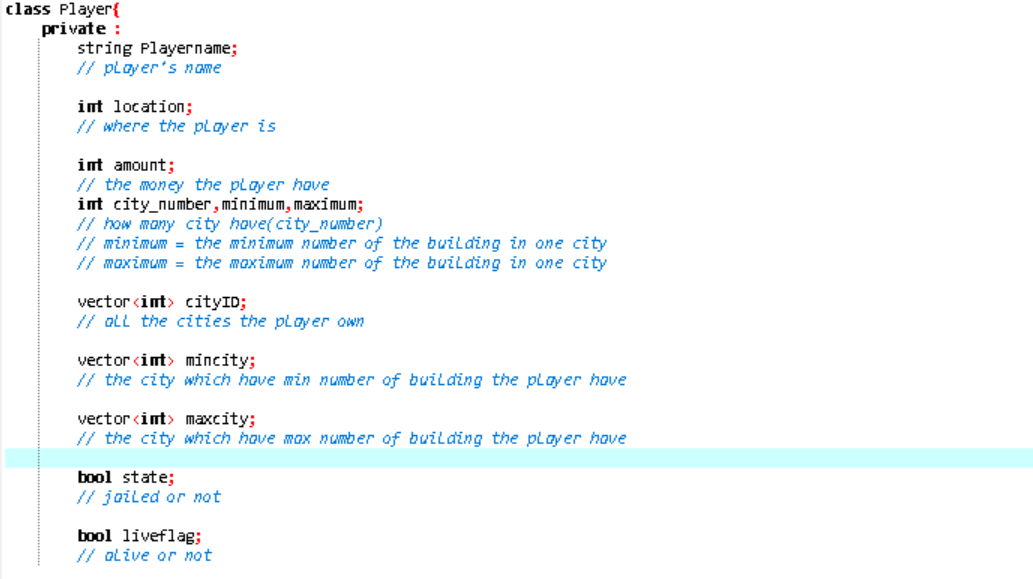
1. **Surface logic**

As we have described the content UI in the above, we just give a brief account here. We have designed 5 different UI ( mainwindow.ui introduction.ui developers.ui gamewindow.ui dicebysound.ui). The users(players) open the program, they will first open the mainwindow.ui. In this surface, they can click on interface introduction.ui, developers.ui and of course--gamewindows.ui. In the gamewindows.ui where the players play the game, they need to dice sometimes, in this case, they will open the dicebysound.ui. As each UI has been described in detail before, we are not going to retell it.

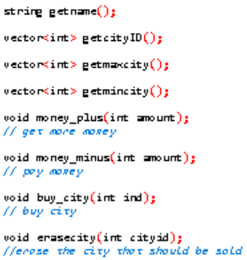
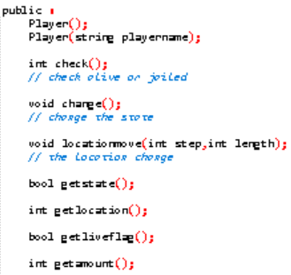
1. **Underlying logic (the relationship between different classes)**

To begin with, we should analysis the demand of the game. Simulating the process of the game, It is easy to to divide it into three different sections. The first is players and the second is map. In the beginning, We design to Implement a class Acount which is included in the class players, However, we find that it is not necessary as sale, purchase and the amount of money owed is unlimited, so we give up to build the class Amount, on the other hand, we extend class players to include the function of the class Amount. Secondly, we design the class Map. Obviously, the map is built by a group of the units. In this section, we should deal with both the unit only have some chances and the unit where represents a city at the same time. So we designed two classes-- the class unit and the class city. Besides, the class unit inherit the class city, so that we can handle at the same time. Last but not least, the class Control have to be designed because the overall process and the interaction of classes should be controlled and adjusted to push the game forward. In this case, the class Control “emerge as the times require”.

1. **Kernel (the introduction of the class)**
2. **Class Player**

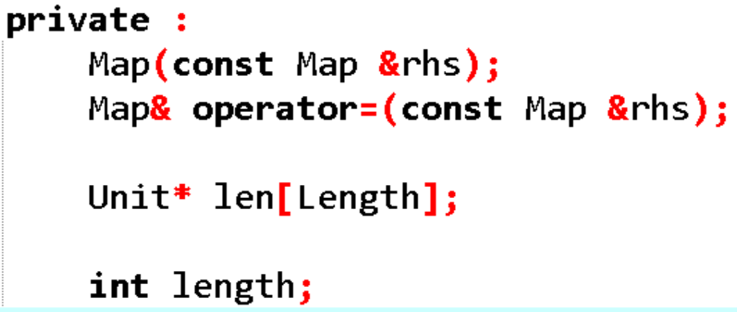


In this class, most of the variables do not need more explanation, we just expound the vector variables. As we have told in rules introduction, the player can master the city so that we should record the city in the name of the player, as the city belong to a player is not fixed, so we design vector variables belong to the class Player to record it.

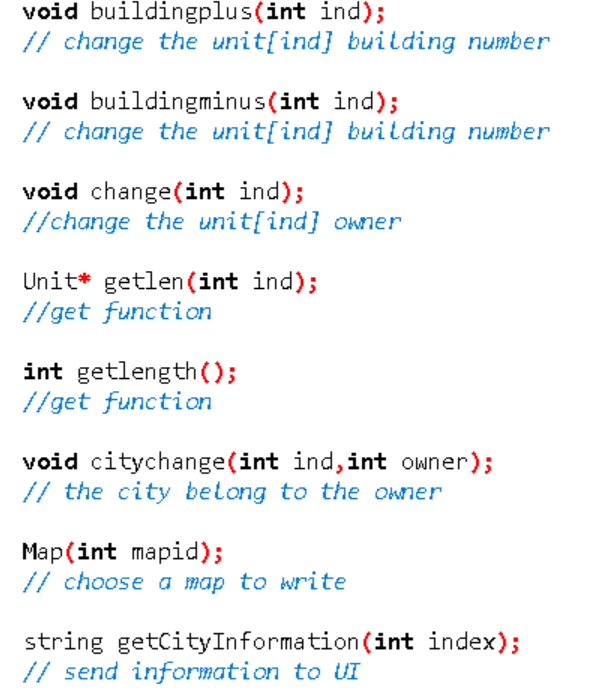


In the class Player, there are some calculate functions be designed so that the player can trade with other players, buy city or building and change the location. What’s more, the class Player still need some button functions, in these functions, we can change the state from the input of the players (users).

1. **Class Map**

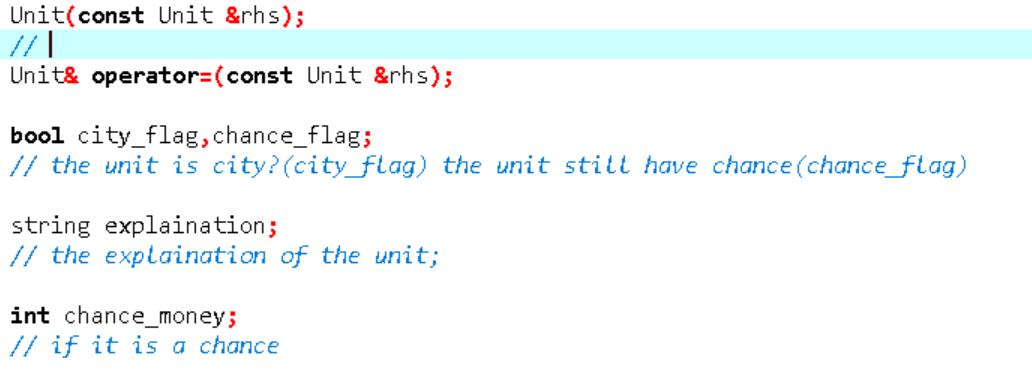


In the class Map, we firstly just use the array made of unit, then we find there some problem if we use the array, for example if we use the get functions, we just copy the array which just have information, which means, we cannot rewrite or modify the Unit array. In this way, we cannot load different map in the function. Therefore, we should change it into the pointer form, so that we can get the address to meet the requirements which need to change the some of the Unit in some functions. By the way, the operator of “=” overloaded to simplify the operation in the functions.

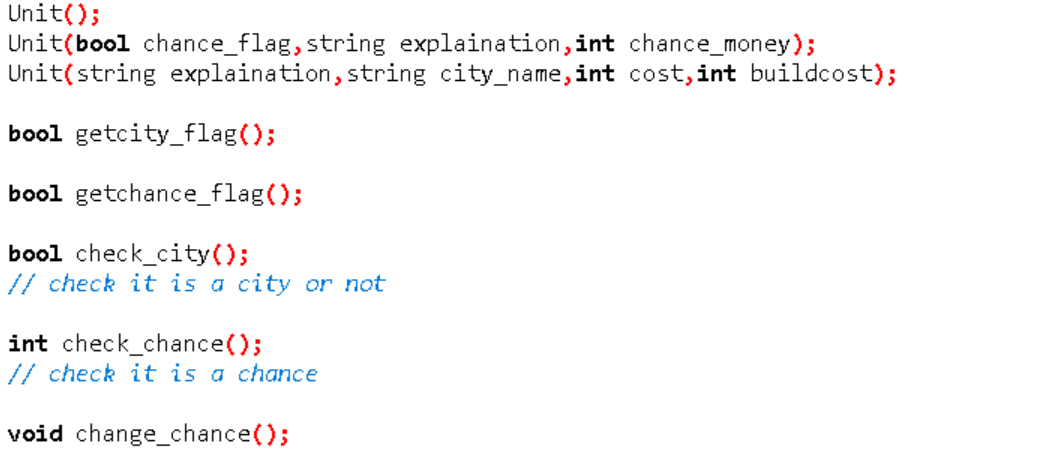


In the class Map, we design these functions so that the other classes can get the information of the Unit and the length. What’s more, as the players can buy or sell the building in the Unit so we design two different functions-- buildingplus and buildingminus due to the requirements. Last but not least, the getCityInformation can get all information of a Unit chosen in the form of string, which can be show in the gamewindow.ui.

1. **Class Unit**

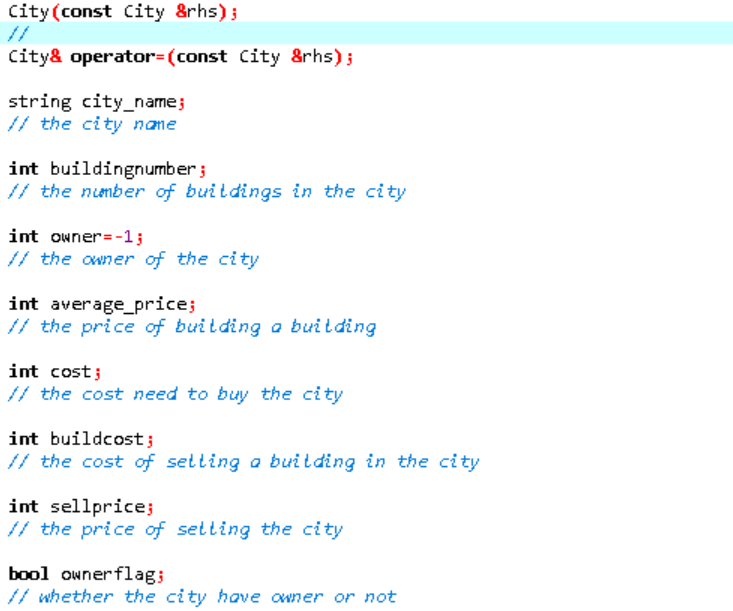


As we should use the class in the class Map, we should specifically handle this class. In this section, as it inherits another class, there are not so much new variables. Besides, here is a variable- explanation which is used for the Unit where have the chance, to introduce the chance to the players.

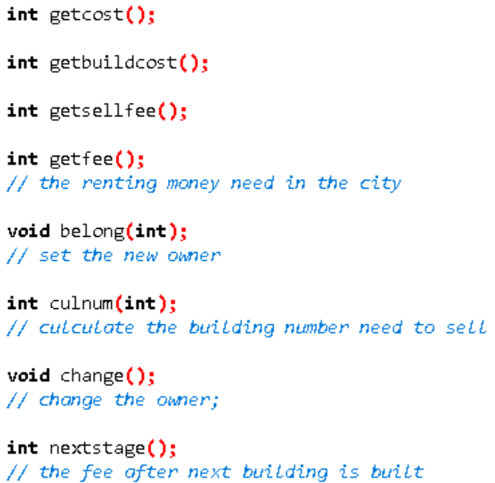
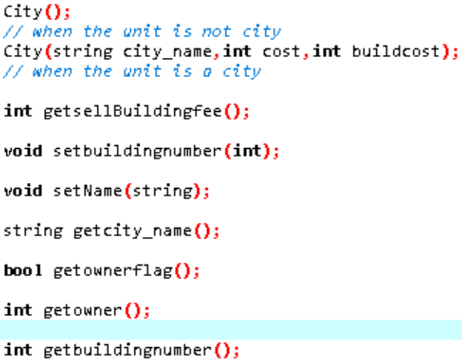


Actually, the unit does not need many functions, most of the functions are get functions (just get the information without handling), on the other hand, there are two different constructors which are used for the chance unit and the city unit.

1. **Class city**

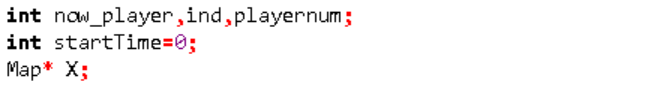


The class city is designed for the city, the variables’ meaning can be seen from the picture. What’s more, the variable named *owner* default to -1 which means the city do not have the owner or not be developed by the owner.

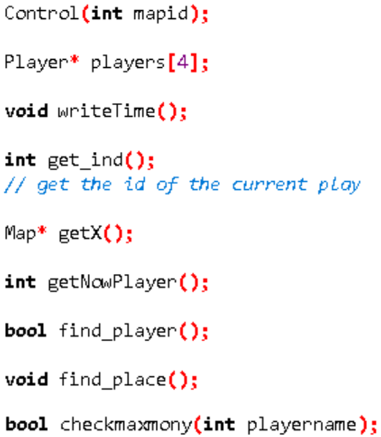
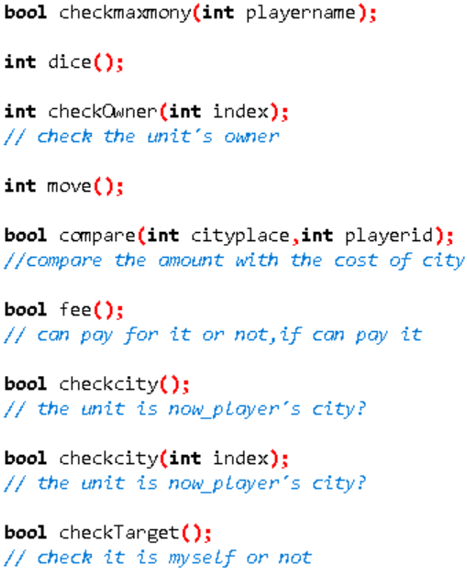


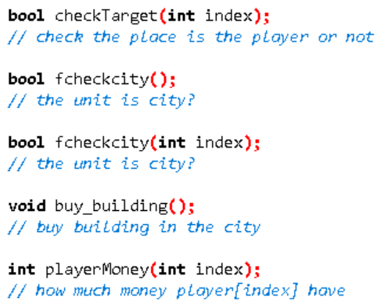
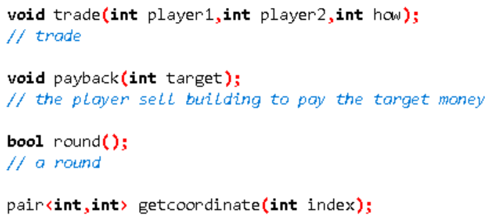
As the class City have lots of messages, we should also have many get functions and set functions. To be honest, a part of the information is calculated in the functions. Besides, we should change the state as the player can buy or sell the buildings in the city, even buy or sell the city.

1. **Class Control**



The class Control does not need many new variables. However, this class need many functions. Both of these are because this class is designed for the interaction among the classes. The variable X in the form Map is a pointer so that we can use the X more freely and easier to debug.

As what we have said, the class Control should have lots of functions for the interaction among the classes. On the one hand, we should have get functions to deliver information to windows classes (which include UI) to show to the users or players. On the other hand, we should have some calculating functions such as find\_ player, find\_ place which calculate to find the information. Then, we have some button functions, when the players input somethings from the UI, our system conduct it on these functions. Last but not least, here is a round function used to adjust and control the whole process.

**The debug procedure**

About the debug procedure of connecting the kernel with ui, we meet many bugs we have not come across.

**Cannot move push button**

At first we try to move the piece but find it no effect, and then find that we forget a line: “ui -> setup Ui(this);”

**The problem of output and input**

The develop of kernel without consider the problem of input and output, so we firstly move the input and output part code to the ui part, and then replace the cout with “ui -> target \_ text -> set Text”, replace the input by the click of push button.

**The problem of implicit copy**

Kernel part passes object by itself which is very easily to implicit copy and cause the problem that sometimes we change some balance on object and in fact the object is the copy one so the change is not applied on the true one. We find this error and then owe it to the implicit copy. We solve it by replace all object with the pointer and then solve this problem.

And the solve of this problem also solve many problems, such as the check part of next which do not work.

**The if – else error**

Such errors are hard to be found as it is not anounce to us. And the result is very serious.

**The strange error**

For example, if we add a word void in parameter on .h file, the function will be failed to be used.

**There are also many unsolved problems.**

**The use of python function is hard to achieve.**

We firstly want to use python function write by ourself, the python code is easy to complete, but when we try to use cpp code to use it, we find that we will fail if we use the library from third party although I have added true library on the .pro file, such as numpy which is very hard to achieve by ourself, and pyaudio to handle .pcm file. which is a pity.