Documentation for CS 101 Autumn Semester 2011-2012 Project

Batch number – 152

Team Members

- Rahul Agrawal - 110040031
- Rohan Das - 110050001
- Rameshwar Gupta - 115320001
- Rohan Gyani - 110040001
- Rahul Nawal - 110010011
- Rohan Sandeep Prinja - 110050011
- Randhavane Tanmay Vitthalrao - 110050010

Project Topic

We chose to design a GUI-based implementation of Monopoly. Monopoly is a highly popular board game that is played typically between two and four players. It teaches the player the fundamentals of business, accounting, entrepreneurship, trading and investing in a fun and engaging way.

The code for the game is written in C++, and the API used is EzWindows.

Why did we choose this topic? Because designing a game of Monopoly is a project that involves using a wide range of the features that C++ has to offer, right from classes to file handling to arrays. Apart from the fact that it is both a challenging and fun game to play, it was also a valuable learning experience for all of us in the team. Keeping this in mind, we chose Monopoly as our topic.

Rules of Monopoly

Monopoly is played on a board between two to four players. Each player starts off with a fixed bank balance (in our case, Rs. 50,000). Players play by turn, and all players start on the START square. When a player's turn comes, they roll a pair of dice and move their piece forward by the sum of the two numbers shown by the dice. Depending on the square they land on, they may have to make some financial decisions.

For example, if they land on a square corresponding to a CITY, they may choose to buy it (if it has no owner yet). If it does have an owner, they must pay a rent to the owner that depends on which city it is and how many houses and hotels (if any) have been constructed in that city. If on the other hand, the player lands on a city that they own, they can choose to build a house or hotel on that city. The costs of constructing houses and hotels vary from city to city. Once the house/hotel is built, the player can charge house and hotel rent (if houses and/or a hotel have been built in that city, otherwise only the basic city rent is charged as rent) from the other player whenever he/she lands on that city. A maximum of three houses can be built in a city, and only one hotel can be built per city.

On the other hand, if they land on a CHANCE/COMMUNITY CHEST square, they gain/lose money depending on exactly which numbers the dice were showing when they reached the chance/community chest square. In some cases, the player may even be sent to jail for certain values of the numbers on the dice! Basically, the chance/community chest squares introduce an element of chance or uncertainty into the game, which makes it a little more exciting.

If a player lands on the INCOME TAX square they have to pay an income tax that depends on how many houses they own. They either pay Rs. 50 per house that they own, or they pay Rs. 500, whichever is less.

The WEALTH TAX square works the same way as the INCOME TAX square, but in this case, we compare the value of Rs. 50 times the number of houses plus Rs. 100 times the number of hotels with Rs. 500, and the player has to pay the lesser amount.

If the player lands on a COMPANY, they may choose to buy it if it doesn't have an owner yet. If it does, they have to pay a rent to the other player. The exact value of the rent depends on which company it is, and whether the other player owns the "paired" company corresponding to that company. Each company has a corresponding "paired company", and if one of the players owns both of these companies, then the other player has to pay a rent that is larger than the usual rent when he/she lands on one of these companies.

The player may also land at a REST-HOUSE (or CLUB-HOUSE) in which case, his/her bank balance is simply incremented (or decremented) by a fixed amount.

If the player lands in JAIL, they are given three options.

1. They can choose to wait it out in jail, that is, they have to miss three of their turns, after which they are allowed to throw the dice and move forward again.

2. Alternatively, they can choose to get out of jail by paying a fine of Rs. 500. In this case, their bank balance is deducted by Rs. 500, and they get to throw the dice again and move forward by the sum of numbers shown on the dice.

3. A third option is that they can throw the dice three times, and they are free to get out of jail if one of the following conditions is satisfied:

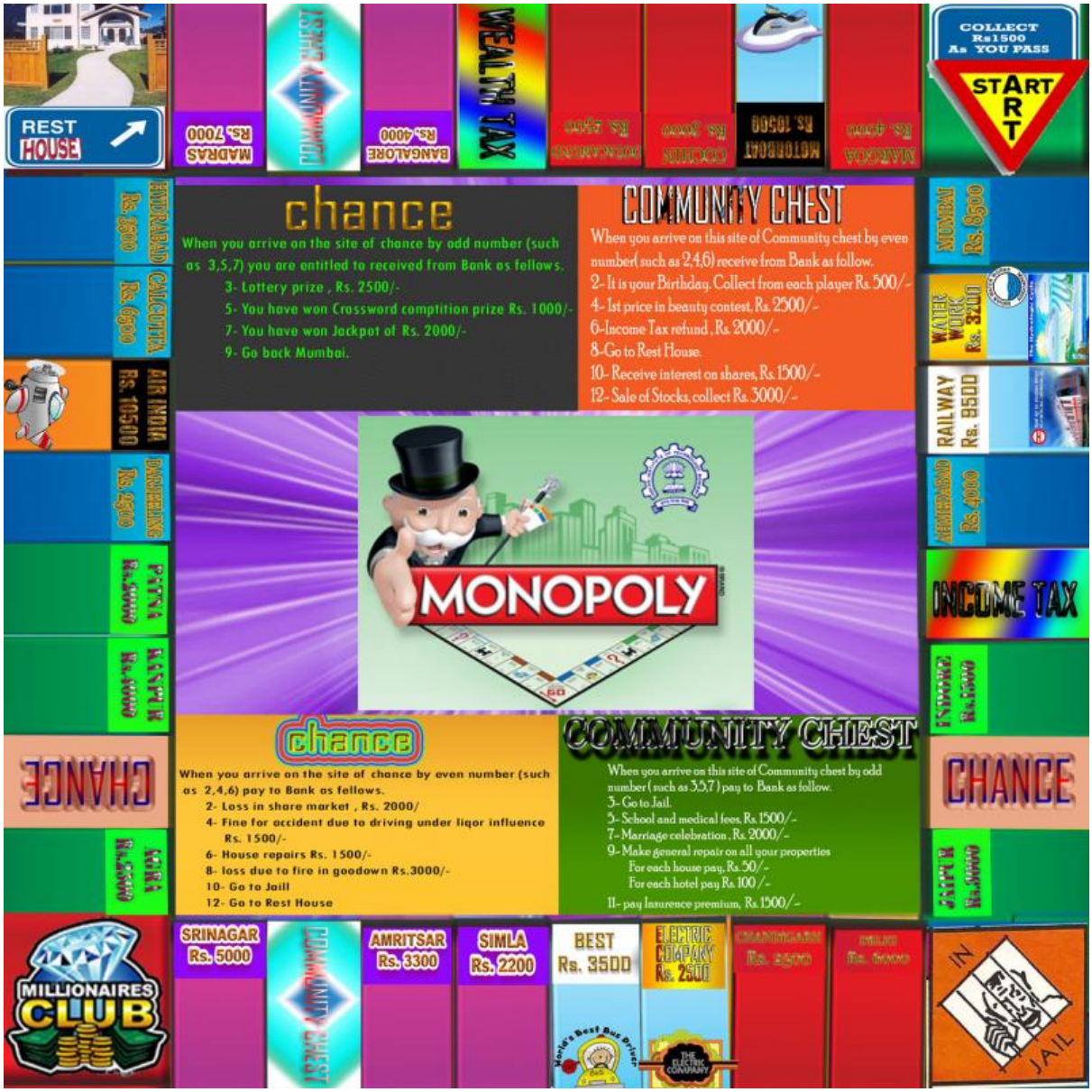
    a. In any one of the throws, the two numbers shown by the dice are the same

    b. In each of the three throws of the dice, the sum of the numbers shown by the dice is the same

    But if neither of the above conditions gets satisfied and the player exhausts all their throws, then the player must pay a fine of Rs. 1000 for failing to satisfy the conditions. They then move ahead by the sum of the numbers shown by the dice in the last throw.

The game ends when only one player is remaining, and that player is the winner. Players quit the game when they become bankrupt, that is, their net assets are zero. If a player is close to bankruptcy, he can mortgage his property, that is, he can relinquish control of his property by selling all of it off at half the price at which he bought it. The property is now considered to be back on the market, that is, it can be bought back by either of the players later on.

The game board layout and structure (designed by team members)



The game board of Monopoly is bascially a loop consisting of 36 squares corresponding to different locations – cities, companies, jail, rest-house, club house, chance and community chest. For the purpose of our project, we have numbered these squares from 0 to 35, with 0 representing the START square, and 35 the very last square of the loop (corresponding to the city of Margoa). The various locations are scattered throughout the 36 squares in an essentially random way.

Classes, Data Members and Methods

The code consists of two sub-parts – class declarations (with function definitions), and the main algorithm. There are three classes declared for this program – the city class, company class, and the player class. The names are self-explanatory. Each class has a number of data members (all private)

and a number of methods (all public) which permit us to inspect and mutate the data members of the object. The access specifiers for the data members and the methods were chosen keeping in mind the data-hiding principle.

The city class – data members and functions

This class consists of the following data members:

- `char city_name[20]` Stores the name of the city in a 20-element array.
- `int city_id` Stores the ID (identity) of the city. There are 20 cities, and we have assigned each an ID ranging from 1 to 20, so as to give them a unique identity.
- `int city_rent, house_rent, hotel_rent` city_rent is the rent charged to a player who is currently located in a city owned by the other player. house_rent is the rent charged for each house of the city, assuming the other player owns it. Similarly for hotel_rent.
- `int house_count` Stores the number of houses constructed in the city by its owner. It can either be 0, 1, 2 or 3. It is initialized to 0 for every object in the city class.
- `int hotel_count` Stores the number of hotels constructed in the city by its owner. It is either 0 or 1, and its default value for each city is 0.
- `int owner` The variable owner is either -1 or 0 or 1. If it is -1, it means that the city is yet to be purchased by one of the players. If it is 0 (or 1) it means that player number 1 (or 2) is its owner.
- `int type` Cities can either be of type 0, 1, 2, 3, or 4. On the game board, this corresponds to the city being either ,or , or , or , or. If a player owns three cities of the same “type” or “color” they can
- `int city_cost` Stores the cost of buying the city. Varies from city to city.
- `int house_cost` Stores the cost of constructing a house in a city. Varies from city to city.
- `int hotel_cost` Stores the cost of constructing a hotel in a city. Varies from city to city.

The city class also has a number of functions:

- `void set_details_of_city(char str[],int id, int c_rent,int hse_rent,int htl_rent,int t,int c_cost,int hse_cost,int htl_cost,int hse_count,int htl_count,int cty_owner);`
- `void display_details_of_city(void);`

- `int get_owner();` `//get owner is either -1 or 0 or 1. if it is -1, no one owns the city yet.`
- `void modify_owner(int new_owner);`
- `void modify_house_count();`
- `void modify_hotel_count();`
- `int return_id();`
- `int return_city_cost();`
- `int return_house_rent();`
- `int return_house_cost();`
- `int return_house_count();`
- `int return_hotel_rent();`
- `int return_hotel_cost();`
- `int return_hotel_count();`
- `int return_type();`

Most of the functions have self-explanatory names. `set_details_of_city` reads data that has been stored in a text file and assigns this data to respective data members of the objects of the city class. `display_details_of_city` outputs the city name, type, city rent, city cost, house rent and cost, hotel rent and cost, and also the owner of the city, if any.

This class consists of 20 objects, since the board has 20 CITY squares.

The company class – data members and functions

Since in Monopoly companies behave very similarly to cities, the data members and functions defined for the company class are almost the same as those for the city class. The company class has some extra data members such as `paired_company_id`, which stores the ID of the company that the given company is paired with, `basic_rent` (rent charged when the other player does not own the paired company) and `paired_rent` (rent charged when the other player also owns the paired company); and also functions like `return_paired_id` (returns the ID of the given company's paired company), `return_basic_rent` and `return_paired_rent`.

This class consists of six objects, since the board consists of six COMPANY squares.

The player class – data members and functions

This class consists of the following data members:

- int player_number
- long int balance
- int house_count
- int hotel_count
- int position
- int city_count
- int turn

player_number is either 0 (representing player 1) or 1 (representing player 2). balance is a long int that is initialized to zero for both the players. house_count is the total number of houses owned by the player in all the cities combined. Similarly for hotel_count. position represents the current position of the player on the board. city_count is simply the total number of cities owned by the player. turn is a variable used specifically in the case when the player is in jail. The value of turn for each player is initialized to zero. When a player lands in jail and he/she chooses to wait three turns, then the turn variable is incremented by 1 each time the player's turn comes. When the value of turn is 3, the player is now free to leave jail as he/she has waited 3 turns as required.

It also consists of the following methods or functions:

- void set_details_of_player(int pn,long int bal,int hse_count,int htl_count,int pos,int cty_count,int p_turn);
- int set_position(int change);
- void modify_balance(int);
- int return_balance(void);
- void modify_city_count();
- int return_city_count();
- void jump(int new_position);
- int return_house_count()
- int return_hotel_count()
- int return_turn()
- int modify_turn(int);
- void modify_house_count()
- void modify_hotel_count();

- void display_details_of_player();

Most of the above functions have self-explanatory names. set_position increments the position of the player by the amount given to it as a parameter. jump is used to literally make the player "jump" from one position on the board to another. For example, in one of the case in the CHANCE square, the player is made to move to another location on the board. To facilitate this abrupt movement we use the jump function, as the set_position function can at most change the position by plus or minus 12 squares only. set_details_of_player is used to read the data corresponding to each player from the main program itself. It reads the data for each player and stores it in the pertinent data member of each player.

Non-class variables and functions

A number of variables and functions that are not part of any of the above classes have been defined and used in our program. Here is a list of all the non-class variables in our program:

Here are the global variables:

- i
- z
- dice_sum
- dice_1
- dice_2

The precise functioning of the variables i and z is specified in the **Program Control Flow** section. dice_sum, dice_1 and dice_2 are variables associated with the dice() function. A complete description of the dice function is given in the **the dice function** section.

Here are the local variables:

- current_position
- d_sum[3]
- tag1
- tag2
- turn
- ch
- x
- y
- temp

current_position simply refers to the position of whichever player's turn it currently is. d_sum[3] is an array of size 3 which stores the values of the sum of numbers displayed on the dice during the time when the player is attempting to break out of jail by throwing the same dice sum thrice.

tag1 and tag2, like d_sum[3] are also associated with the jail function. They are by default 0, and they take the value 1 when the corresponding jail condition is met. For a more in-depth description of the functioning and use of these variables, see the **the jail sub-algorithm** section.

The turn variable is also associated with the jail sub-algorithm.

ch stands for 'choice'. It is a char variable which stores the player's response when he is asked whether he wants to buy a city or company.

x and y are variables that act as array indices for the arrays cty[] and cmpny[] respectively. x runs from 0 to 19 and y runs from 0 to 5. They are used in order to check exactly which city (or company) it is that the player has landed on. The main algorithm is structured such that after all the various possibilities for a location have been exhausted (e.g. CHANCE, REST-HOUSE, JAIL etc.) we add an else statement. If the current location is neither a CHANCE nor COMMUNITY CHEST nor CLUB HOUSE nor REST-HOUSE nor JAIL nor INCOME TAX nor WEALTH TAX by definition it must be either a CITY or a COMPANY. In order to check exactly which city (or company) it is, we run the index variables x and y in their respective limits, making use of of an if statement and the return_id member functions for both the city and company classes.

temp is simply a variable that stores the value (cty[x].return_house_count())*(cty[x].return_house_rent())+(cty[x].return_hotel_count())*(cty[x].return_hotel_rent()). This, as one can guess, is the total rent that one must pay to the owner of the city assuming some houses and/or a hotel have been constructed.

Program Control Flow

Positions of the game board

The game board, as described above consists of 36 squares arranged in a loop with each square corresponding to a particular location on the board. 20 of the locations are cities, 6 of them are companies, and there is one square each corresponding to jail, rest-house, club-house, start, income tax and wealth tax. There are two squares each for chance and community chest. We have numbered the locations on the board from 0 to 35, where 0 corresponds to START and 35 corresponds to the last location in the loop, which in our case is the city of Margoa.

Main Algorithm

Most of the main algorithm of the program is enclosed in an while loop that runs as long as two conditions are simultaneously satisfied: both players have a positive bank balance, and also, the value of the ch2 char variable (functioning of the ch2 variable will be described shortly) is either 'n' or 'N'. The program starts off by reading the data stored in the text files company.txt and city.txt, and storing the data in the appropriate class data members. For example, it stores the values of the city name, city cost, city rent, house cost, house rent and other data members for the city and stores them in the respective city object.

It then reads the data directly for the players and displays the details for each player using the public player class function display_details_of_player.

We use a variable z to determine whose turn it is. This variable is incremented after each iteration of the while loop, and the variable i (defined as i=z%2) refers to the player whose turn it is. i=0 means that it is player 1's turn, and i=1 means it is player 2's turn.

The current bank balance of the player is displayed, and then the dice function is called. The player is moved forward by the sum of numbers appearing on the dice, and the position of the player is incremented by passing an appropriate parameter to the set_position function for the player object. Various cases are then considered using a set of if statements in order to determine what action the player should take.

Whenever the bank balance of a player is to be changed (either increased or decreased) we make use of the player class public function modify_balance. The parameter passed to modify_balance is simply an integer representing the algebraic change in the player's bank balance. If, for example, we pass -500 to the modify_balance function, the player's bank balance is decreased by Rs. 500 (or alternatively, increased by Rs. Minus 500)

For example, if the position of the player is now 7 or 20, that is, the player is on a CHANCE square, then by looking at the value of dice_sum, the player gains/loses some money or is sent to jail/rest-house etc. Since there are as many as 12 different cases to be considered for both CHANCE as well as COMMUNITY CHEST squares, we use the switch statement of C++.

In the same way, we consider the different cases of when the player is on JAIL, CLUB-HOUSE, REST-HOUSE, INCOME TAX or WEALTH TAX squares and accordingly take action. See the rules of the game for a description of the actions taken when the player is on the INCOME TAX, JAIL and WEALTH TAX squares.

The REST-HOUSE square simply adds Rs. 100 to the bank balance of the player, whereas the CLUB HOUSE deducts Rs. 100 from the player's bank balance.

Another case considered is when the player passes the START square. In that case, according to the rules of the game, the bank balance of the player is incremented by Rs. 500. How do we check whether the player has passed the START square?

Recall that we had numbered each location on the board starting from 0 and going up to 35. If a player crosses the START square, we can surely conclude that his position after moving forward by the appropriate number of paces is less than the sum of the two numbers appearing on the dice. For example, say the player is on location 31 and he throws a 8. His new position will be 3 (because 31 +

8 =39 and 39 = 3 (mod 36)). 3 is less than 8. This is in fact obvious, since the sum of numbers appearing on dice = new position + 36 – old position. Since old position is between 0 and 35, 36 – old position is positive, and so new position + a positive quantity = sum of numbers, so sum of numbers is greater than the new position. This is the condition that we use to check if the player has passed the START square.

Here is the order in which we check for different locations on the game board, along with a quick summary of the actions performed there:

1. Chance, which consists of twelve sub-cases that are dealt with using the switch statement of C++.

2. Community Chest, which, like Chance, also consists of twelve sub-cases dealt with similarly.

3. Jail, which consists of three sub-cases (dealt with using if statement) – player pays a fine of Rs. 500, player waits three turns before his next move, or lastly, player attempts to either get a doublet or the same dice sum in three throws in order to escape, and pays a fine of Rs. 1000 if unable to do so.

4. Club House – player's bank balance decremented by Rs. 100.

5. Rest-house – Rs. 100 deposited in player's bank balance.

6. Start – Rs. 500 deposited in the player's bank balance when he lands on or crosses the Start square.

7. Income tax – player pays income tax determined in part by the amount of property (number of cities) that he owns. Player pays either Rs. 500 or Rs. 50 for each city that he owns, whichever is less.

8. Wealth tax – very similar to income tax. In this case, the amount of property is measured by the number of houses and hotels owned by the player and not the number of cities as was the case in income tax.

9. Company – player is given the option to buy the company (if it doesn't have an owner yet), or is made to pay rent if the company happens to be owned by the other player. The rent charged goes up if the other player is also in possession of the paired company.

10. City – similar to company. If the city has no owner, player is given a chance to buy it. If the other player owns it, the player must pay rent determined by the number of houses and hotels already built in the city. If the player owns the city, and there is scope for construction, he is asked if he wants to construct a house/hotel in the city (maximum of three houses and one hotel allowed).

At the very end of the while loop, the player is asked if he wants to quit. If his response is 'y' or 'Y', the control passes out of the while loop, since one of the conditions of the while loop was that ch2 (the variable we use to read the user's choice o continue the game) should be equal to 'n' or 'N'. After the while loop, we then check for whether the response to ch2 was 'y' or not. If it was 'y', then the other player is declared the winner of the game, since the player has resigned. But if the

response to ch2 was not 'y', then the very fact that the loop has been broken out of means that the condition that bank balance of the player is positive has not been satisfied. In this case, the player's property is mortgaged.

The mortgaging process has been simplified slightly in our implementation of the game. In the main program, we check for bankruptcy with an if statement, and if the player is found to be bankrupt, his properties are mortgaged off, or in other words, they are put back on the market with the player receiving half the market price of each city and company that he previously owned.

After mortgaging off the property of the player, we check whether his bank balance is negative or not. If it is negative, then it means that the player's bank balance did not become positive even after the mortgaging, and so the player is by definition bankrupt. That means the other player has won the game.

The jail sub-algorithm

When the player lands in jail, he has three choices. He can either wait three turns without making a move and then continue playing. Or, he can pay a one-time fine of Rs. 500 and continue playing. Finally, he can attempt to leave jail by throwing the two dice three times in a row. If at any point he gets a doublet (both dice showing the same number) or the dice sum is the same in all three throws, then he is free to leave the jail, and he is moved forward by an appropriate amount. But if he fails to get a doublet, and the dice sum is not the same for all three throws, he has to pay a fine of Rs. 1000, and in the next turn, he gets to play.

The case where he pays a fine of Rs. 500 has been dealt with by using the modify_balance function of the player class. The case where the player has to wait out three turns in jail is dealt with by making use of the turn variable. See **The player class** section for a description of the way turn is used to ensure that the player misses three turns.

In the third case, when the player has to throw the dice three times and either get a doublet or get the same dice sum each time, we make use of two variables tag1 and tag2. These two variables serve to identify which one of the conditions (if any) for getting out jail is met. tag1 corresponds to getting a doublet, and tag2 corresponds to getting the same dice sum each time. Both tag variables are initially set to zero. An integer array d_sum[3] is used to store the value of the dice sum for each one of the three throws of the dice. We run a for loop for a variable j that goes from 0 to 2. In each iteration of the loop, we check whether a doublet has been obtained, and if it has, we set tag1 = 1 and break out of the loop, else we just continue.

Then, we put an if condition that asks whether tag1 is zero AND all three entries of the d_sum[] array are equal. If yes, then that means that the first condition has not been satisfied, but the second one has. Then, firstly, the value of tag1 is set to 1, and secondly, the player's turn variable is reset to zero by multiplying -1 with the output of the return_turn player class function, and adding the resultant to the current value of the turn for the player.

We then have another if condition that checks whether the values of either one of tag1 and tag2 are 1. If yes, that means that at least one of the conditions has been satisfied for leaving jail, and the

player is moved forward by a number of squares equal to the sum of numbers shown on the dice in the last dice throw.

The dice function

The dice function simulates the throwing of two dice. It returns the value of the sum of the numbers shown by the two dice. To simulate the throwing a die we need to use a randomizing function such as the rand() function found in the cstdlib library of C++. The output of the rand() function is a long integer, which we divide by 6 and add 1 to obtain a random number lying between 1 and 6. We also provide the srand() function with a suitable seed value, which conventionally is the output of the time() function (which is found in the ctime library).

Because the typical computer today processes code so rapidly that the time gap between a few lines of code in C++ is typically much, much less than a second and also, because the return value of the time() function is a long integer measuring the number of seconds passed since a fixed date, it is not enough to simply call the rand() function twice. If the first rand() function is called at time equal to t (say), then the second rand() function would be called at, say, t plus 0.000001 seconds. But the seed value wouldn't have changed at all, since time()'s output is the number of seconds (the decimal part is truncated). So then, the output of both rand() functions would be the same, with the result being that the dice function would always lead to throws like (1,1), (5,5), (3,3) and so on! This is a huge design flaw that we circumvented by calling the srand() function again after calling rand() once, and this time giving time() + 1000 as the parameter to the second srand(). This ensures two totally random dice throws.

Graphical component of the code

In our program we have made use of the EzWindows API. A brief listing of all the windows and images used in the program is given below.

Windows used in the program:

Each window used in the program is an object belong to the pre-defined SimpleWindow class in EzWindows.

- Welcome Window – it displays a welcome message to the player and instructs them to go to the Terminal and press 1 in order to begin playing the game.
- Monopoly – it is the window that is displayed when the dice throw is taking place. It displays two bitmap images showing the dice and the numbers obtained when the dice() function is called.
- Game Board – the window where the actual game board used for playing the game is displayed. It also displays the two pieces used that represent the position of the players on the board, as well as the houses and hotels constructed during the progress of the game.

Bitmaps used in the program:

- BitMap Dice_text1;
- BitMap Dice_1;
- BitMap Dice_2();
- BitMap House[60];
- BitMap Hotel[20];
- BitMap Player[1];
- BitMap Game_Board;
- BitMap Welcome_Image

Dice_text1 is displayed in the Monopoly window. It basically acts as a background image during the throwing of the dice.

Dice_1 and Dice_2 are also displayed in the Monopoly window, and as the names suggest, they correspond respectively to the first and second dice. The number displayed on them is determined by the dice() function described earlier in this document.

House[] is simply an array of identical images, and its size is 60, as there are a maximum of 3 houses per city, and there are 20 cities. Similarly, Hotel[] is an array of 20 bitmaps, since in the case of hotels, we can only construct at most one hotel for a city

Game_Board is the bitmap corresponding to the actual game board. It is displayed in the Game Board window.

Welcome_Image has a self -explanatory name. It is displayed in the Welcome window.

We have also defined an array of objects of the pre-defined Position class of EzWindows. Each element of this array stores the coordinates of the positions where the components of the game board are located. For example if the position coordinates of a location that the player has to move to after throwing the dice are (x,y) then that is the position that the player's piece will be moved to.

Thank you