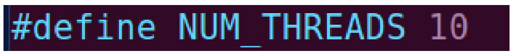
Report

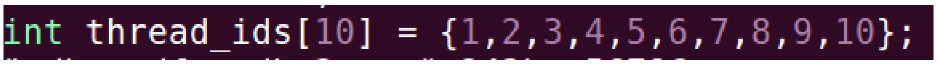
Li Huayue 118010138

1 The thought of design:

After observation, I find there are 9 logs (the length of the river is 9) and 1 frog in total. In order to achieve and manage the program using multiple thread, I create 10 threads in total and give index to each of them so that I can control the work of every thread in functions with their index.

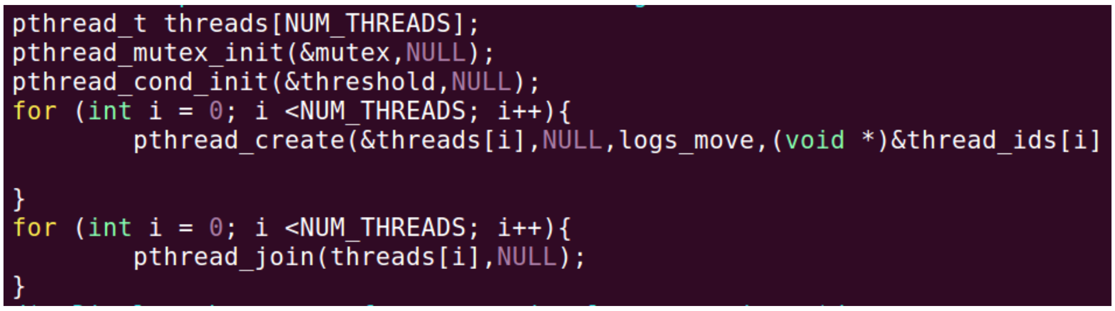


```
#define NUM_THREADS 10
```



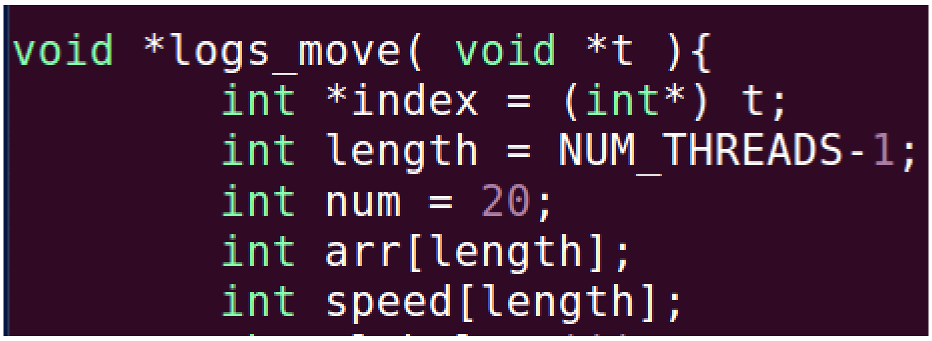
```
int thread_ids[10] = {1,2,3,4,5,6,7,8,9,10};
```

In the main function, I first draw the river map in the terminal with the code of source provided. Then I create all the threads, the mutex and the condition and let all the threads to execute the function "logs_move" with their index as parameters. I also use the "pthread_join" to let the main thread wait for the end of all the child threads.



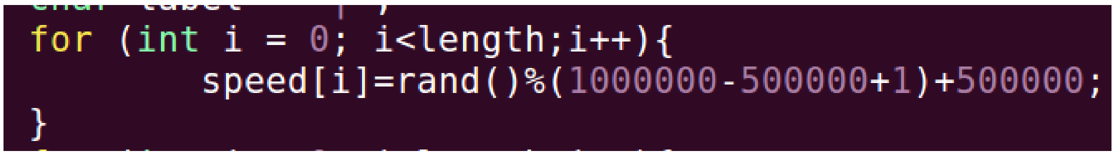
```
pthread_t threads[NUM_THREADS];
pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&threshold,NULL);
for (int i = 0; i <NUM_THREADS; i++){
        pthread_create(&threads[i],NULL,logs_move,(void *)&thread_ids[i]

}
for (int i = 0; i <NUM_THREADS; i++){
        pthread_join(threads[i],NULL);
}
```

The logs_move execute the work of not only the move of logs but also the move of the frog. The index variable is used to store the index of the corresponding thread. The num variable is used to stored the length of each log (in the program, all the logs are of the same length but in the bonus program, I set all the logs of random lengths which means the length of all the logs may be different). The arr array is an array used to store the initial position of the "=" leftmost of each log. To achieve the random different speed of different logs, I use an array "speed" to store the speed of each log.

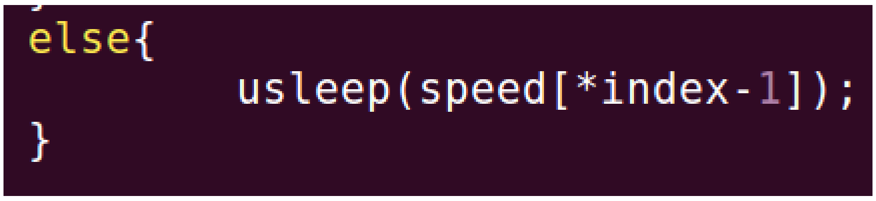


```
void *logs_move( void *t ){
        int *index = (int*) t;
        int length = NUM_THREADS-1;
        int num = 20;
        int arr[length];
        int speed[length];
```

The way I set all the logs in different random speed is shown as follows:

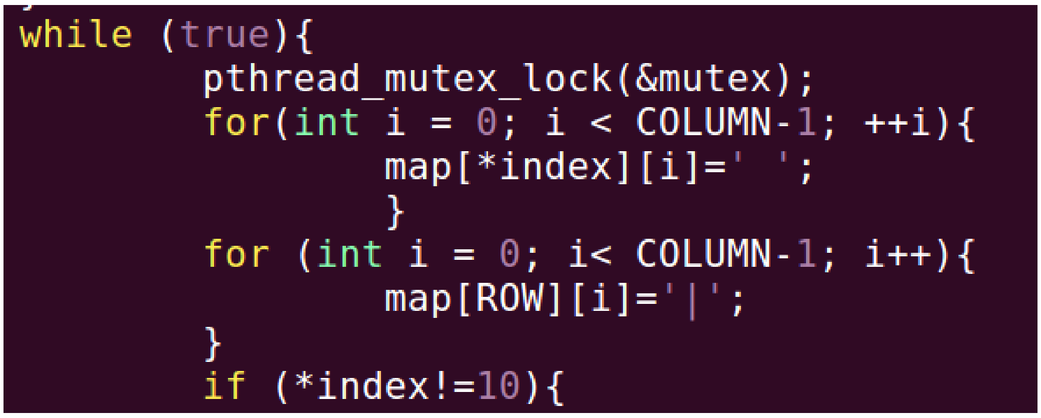


```
for (int i = 0; i<length;i++){
        speed[i]=rand()%(1000000-500000+1)+500000;
}
```



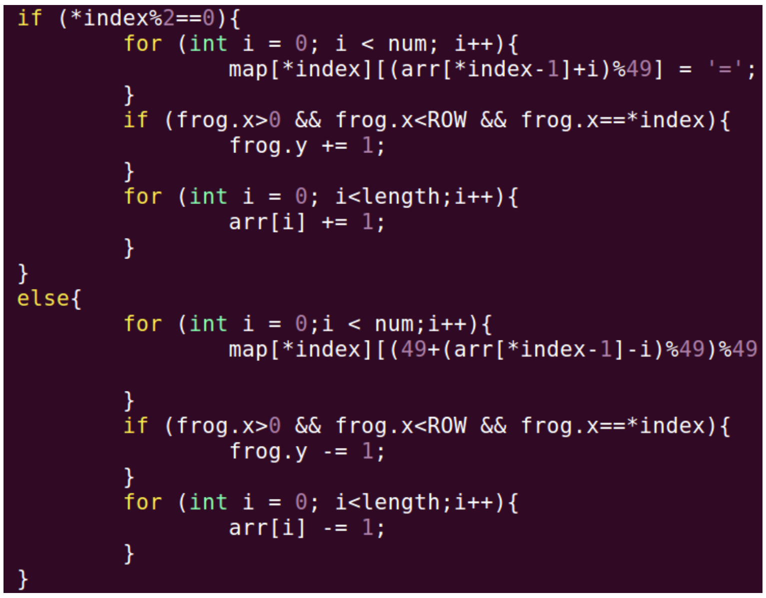
```
else{
        usleep(speed[*index-1]);
}
```

When all the work of preparation done, it is time to make the continue move of logs and the move of frog. There is a "While (true)" structure in the function and there is a "if" to judge the thread is of frog or log. If the index is 10, it will execute the work of the frog while if the index is not 10, the while will execute all the work of logs.



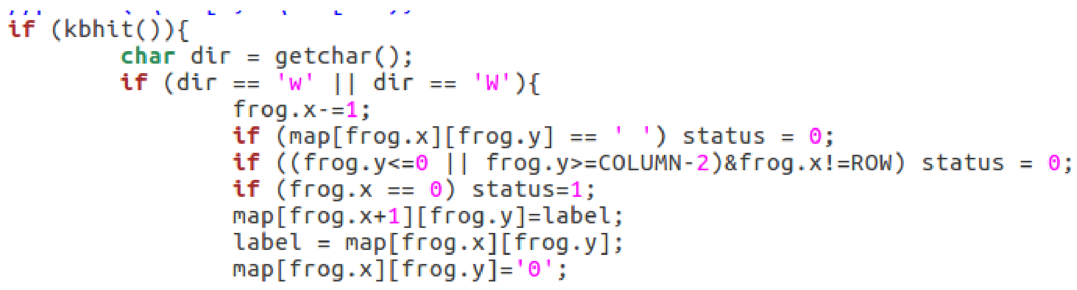
```
while (true){
        pthread_mutex_lock(&mutex);
        for(int i = 0; i < COLUMN-1; ++i){
                map[*index][i]=' ';
                }
        for (int i = 0; i< COLUMN-1; i++){
                map[ROW][i]='|';
        }
        if (*index!=10){
```

In the part of work of logs, to achieve the move of logs, I first write the logs and then make all the index plus or minus 1 then after every “usleep” end, we can see the move of logs. What is more, to make the logs move from left to right or right to left staggerly. I set the logs of even index move from left to right and set the logs of odd index move from right to left. To make the relative speed to be 0 between the log and the frog when the frog is static on that frog. I set the position of the frog move with the move of the log.



```
if (*index%2==0){
        for (int i = 0; i < num; i++){
                map[*index][(arr[*index-1]+i)%49] = '=';
        }
        if (frog.x>0 && frog.x<ROW && frog.x==*index){
                frog.y += 1;
        }
        for (int i = 0; i<length;i++){
                arr[i] += 1;
        }
}
else{
        for (int i = 0;i < num;i++){
                map[*index][(49+(arr[*index-1]-i)%49)%49

        }
        if (frog.x>0 && frog.x<ROW && frog.x==*index){
                frog.y -= 1;
        }
        for (int i = 0; i<length;i++){
                arr[i] -= 1;
        }
}
```

In the part of work of the frog, I use the “kbhit” which is provided by the source to get the input of my keyboard. For the move of the frog, there are four kinds of input of keyboard: “w/W”, “a/A”, “s/S” and “d/D” (my program can recognize the capital and lower-case letter of the input): use the “w/W” as an example:



```
if (kbhit()){
        char dir = getchar();
        if (dir == 'w' || dir == 'W'){
                frog.x-=1;
                if (map[frog.x][frog.y] == ' ') status = 0;
                if ((frog.y<=0 || frog.y>=COLUMN-2)&frog.x!=ROW) status = 0;
                if (frog.x == 0) status=1;
                map[frog.x+1][frog.y]=label;
                label = map[frog.x][frog.y];
                map[frog.x][frog.y]='0';
```

After each move of the frog, the program will judge the status of the game, If you lose or win, the status will store the information and then end the program to show the information “win/lose”. There is also a special input of the keyboard “q/Q” which means to quit the game:

```
}else if(dir == 'q' || dir == 'Q'){
        map[frog.x][frog.y]='0';
        status = -1;
}
```

No matter the user input the move of frog or not, the thread of frog will always clear the terminal and draw the case of the game in the terminal again after every usleep of the frog thread:

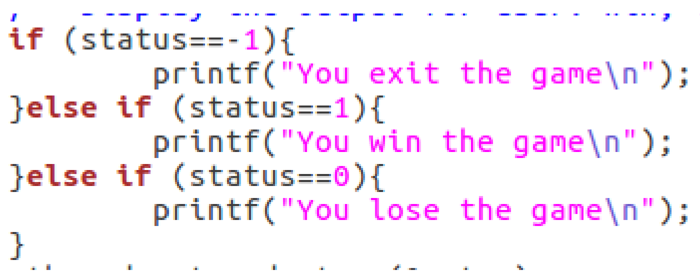
```
printf("\033[0;0H\033[2J");
for(int i = 0; i <= ROW; ++i){
        puts( map[i]);
}
```

The usleep of the frog and the logs are shown as follows:

```
if (*index == 10){
        usleep(100000);
}
else{
        usleep(speed[*index-1]);
}
```

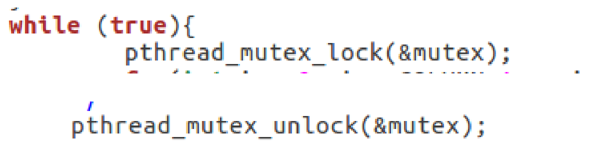
To make the frog move more flexible than the logs, the usleep time of the frog is much less than all the logs. And every loop, the program will judge the value of the status, if the value of the status achieves the requirements, the game will end and print the information of the result:

```
if (status==-1 || status == 0 || status ==1){
        printf("\033[0;0H\033[2J");
        pthread_exit(NULL);
}
```



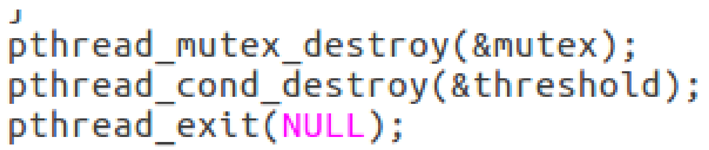
```
if (status==-1){
        printf("You exit the game\n");
}else if (status==1){
        printf("You win the game\n");
}else if (status==0){
        printf("You lose the game\n");
}
```

I use only a mutex lock in my program (in fact there are two mutex lock in my program, because if we split the “if and while” structure into two functions, there are two “while” structure in actual, so the number of mutex lock is 2 in fact.):



```
while (true){
        pthread_mutex_lock(&mutex);

    pthread_mutex_unlock(&mutex);
```

I put the mutex lock at the beginning of the “while” structure where is the begin of the game and unlock the mutex after all the works of the frog and the logs where is the almost the end of every loop. I put the mutex lock here to make sure that the work of the frog thread and every log thread will not interfere because there are some variables are shared among the threads. If I use the mutex lock here, the interfere will disappear. Finally, after the end of the game, all the information of the threads will be released.

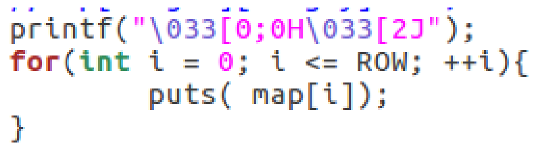


```
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&threshold);
pthread_exit(NULL);
```

Above all are all the thoughts of my design and the position and reason of the mutex lock.

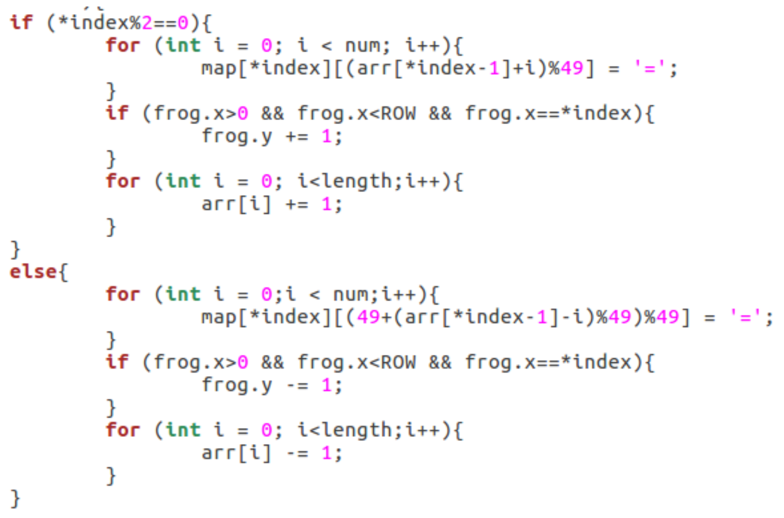
2 The problems met and the solutions:

The first difficult problem I have met is that I first put all the work of set “bank” and draw the map at the end of the "while" structure which means that all the threads will clear the terminal and draw the map after every usleep. Because there is a difference among all the usleep of the threads, when I run the program, the map will "flicker". In order to solve the problem, after observation, I find that the usleep of the frog thread is lowest, so I put the work of clear the terminal and draw the map at the end of the frog work thread which means that only the frog thread will clear the terminal and draw the map after every usleep of it. Finally, this problem is solved.



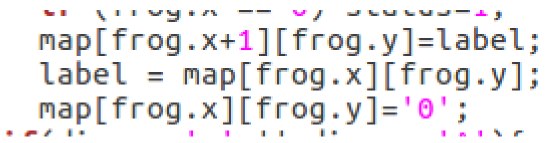
```
printf("\033[0;0H\033[2J");
for(int i = 0; i <= ROW; ++i){
        puts( map[i]);
}
```

The second difficult problem I have met is that after observation, I find that the assignment requires when the logs reach the end of the river, they will output from the other side to create a loop. I use the index plus and minus of the logs to make the logs move, so the value will finally exceed the max index of the river or become less than 0. To overcome this problem, I use the "%" operation to make sure that when I draw the logs, the index is always within the range of the [0, max_index of the river] interval. Finally, this problem is solved.



```
if (*index%2==0){
        for (int i = 0; i < num; i++){
                map[*index][(arr[*index-1]+i)%49] = '=';
        }
        if (frog.x>0 && frog.x<ROW && frog.x==*index){
                frog.y += 1;
        }
        for (int i = 0; i<length;i++){
                arr[i] += 1;
        }
}
else{
        for (int i = 0;i < num;i++){
                map[*index][(49+(arr[*index-1]-i)%49)%49] = '=';
        }
        if (frog.x>0 && frog.x<ROW && frog.x==*index){
                frog.y -= 1;
        }
        for (int i = 0; i<length;i++){
                arr[i] -= 1;
        }
}
```

The largest problem I have met in this assignment is that my frog “0” appear to exist 2 or more “0” in a very little time. Although the time is very little and finally there will exist only 1 “0”, I think I need to overcome this problem or my game will look strange. After observation, I find the bug is that, the draw of the logs is made in the part of the logs work while the draw of the frog is made in the part of the frog work. Because the usleep of the frog is much less than any one of the usleep of the logs. When the frog thread draw the “0”, the log thread still does not draw the “=”, therefore, appear to exist 2 or more “0” in a very little time. To overcome this problem, I create a variable to store the label (in fact, most of the time, the label is “=”) which is taken placed by the frog from the last step. When the frog move to the next position, the frog thread will recover the label of the last position of the frog. Finally, this problem is solved.

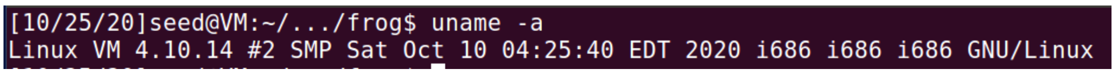


```
map[frog.x+1][frog.y]=label;
label = map[frog.x][frog.y];
map[frog.x][frog.y]='0';
```

Above all are all the problems that I have met and all the solutions.

3 The environment of running the program and the steps to execute the program:

The environment of running the program is:



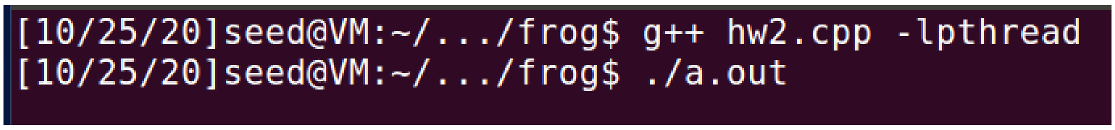
```
[10/25/20]seed@VM:~/.../frog$ uname -a
Linux VM 4.10.14 #2 SMP Sat Oct 10 04:25:40 EDT 2020 i686 i686 i686 GNU/Linux
```

The steps to execute the program:

(1) Enter the folder containing the “hw2.cpp” and open the terminal in that folder.

(2) Enter “g++ hw2.cpp -lpthread” in the terminal.

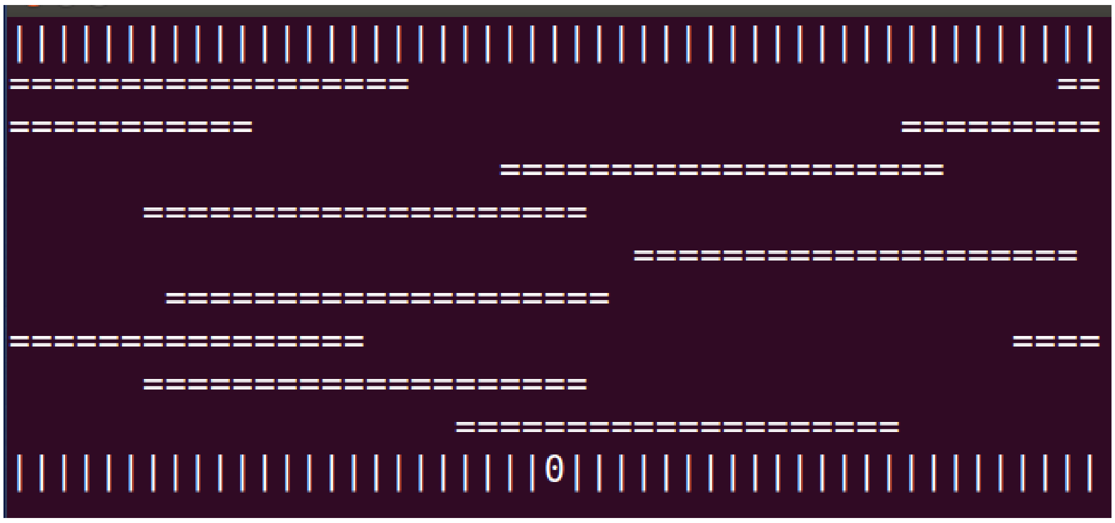
(3) Enter “./a.out” in the terminal.



```
[10/25/20]seed@VM:~/.../frog$ g++ hw2.cpp -lpthread
[10/25/20]seed@VM:~/.../frog$ ./a.out
```

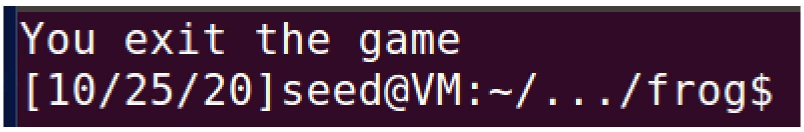
4 The output of the program:

The screenshot of the game:



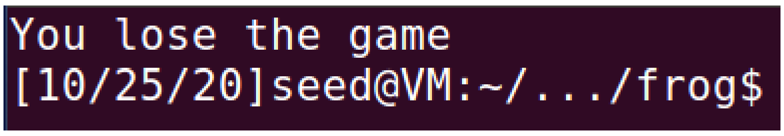
```
|||||||||||||||||||||||||||||||||||||||||||||||||
==================                             ==
===========                             =========
                      ====================
      ====================
                            ====================
       ====================
================                             ====
      ====================
                    ====================
||||||||||||||||||||||||0||||||||||||||||||||||||
```

If you enter “q/Q” when the game is running:



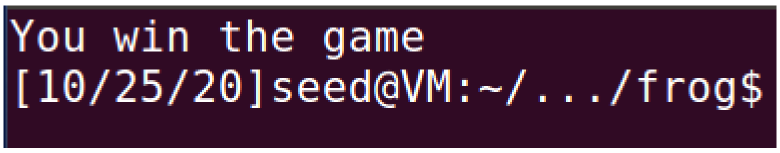
```
You exit the game
[10/25/20]seed@VM:~/.../frog$
```

If you lose the game (the frog drops into the river or reachs the side of the river on the log):



```
You lose the game
[10/25/20]seed@VM:~/.../frog$
```

If you success to make the frog reach the other bank of the river:



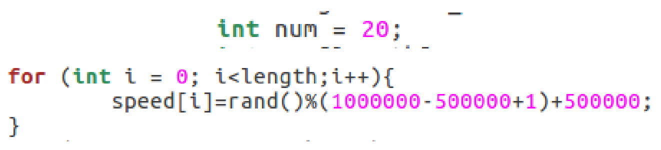
```
You win the game
[10/25/20]seed@VM:~/.../frog$
```

5 Some hints:

(1) According to the rule of the game, the game will lose if the frog reach the left or

right side of the river on the logs, but what you should pay attention to is that if you reach the left or right end side of the bank, the game will not lose in the program.

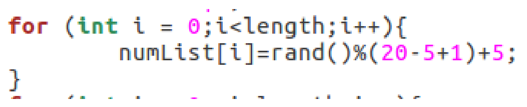
(2) To make it easy to test the program, I set the length of the logs a little long and the speed of the logs a little slow. In fact, when you run the program, you can set the length of the logs and the speed of the logs as you like:



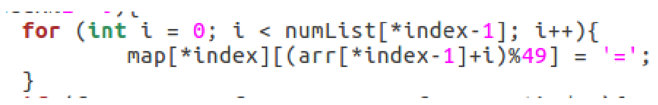
```
                int num = 20;

for (int i = 0; i<length;i++){
        speed[i]=rand()%(1000000-500000+1)+500000;
}
```

6 Bonus:

For the bonus part of this assignment, I implement the second and the third requirements of the bonus. In my bonus program, the length of all the logs are random within a interval:

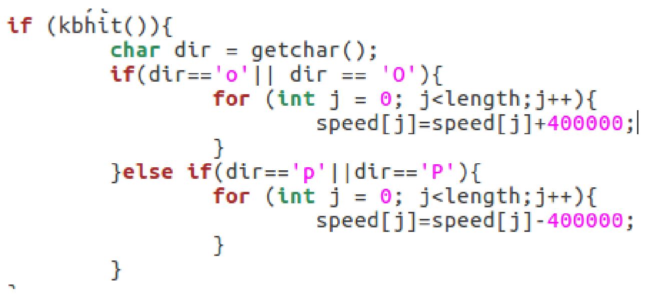


```
for (int i = 0;i<length;i++){
        numList[i]=rand()%(20-5+1)+5;
}
```



```
for (int i = 0; i < numList[*index-1]; i++){
        map[*index][(arr[*index-1]+i)%49] = '=';
}
```

What is more, to make the change of the speed of the logs, I put two new letters “o/O” and “p/P” on the keyboard to control the speed of all the logs. When you enter “o/O”, the speed of all the logs will decrease and when you enter “p/P”, the speed of all the logs will increase.



```
if (kbhit()){
        char dir = getchar();
        if(dir=='o'|| dir == 'O'){
                for (int j = 0; j<length;j++){
                        speed[j]=speed[j]+400000;
                }
        }else if(dir=='p'||dir=='P'){
                for (int j = 0; j<length;j++){
                        speed[j]=speed[j]-400000;
                }
        }
```

7 What did I learn from this assignment:

In fact, this is my first time to use multiple threads to write a program especially a game. I have learnt how to use multiple threads to write a program and how to handle the conflict among the threads using mutex lock. I also learnt how to make the terminal correspond to the input of the keyboard. I have met a lot of problems and overcome them one by one. This program is interesting and I get the fun of writing a game using C++. I have learnt a lot from this assignment and thank you very much.