Programming in C++: Problem Set 1

Sam Tanner: Candidate No.253074

October 26, 2021

1 Intro

In this problem set I was tasked to re-create the game Craps in C++ through breaking the problem down into its components and investigating each respective component. These were random number generation of floats, simulating two six faced dice using said number generation and then using both components to simulate a game of Craps. To do so accurately I had to evaluate that my code functioned correctly and as such show each component to give the expected results.

Therefore I demonstrated that the variance of the randomly generated float between 0 and 1 was the same as that expected mathematically; that it's variance tends to $\frac{1}{12}$ as the number of numbers generated tends to infinity. For the rolling of two dice I looked at the distribution of values for a range of iterations and ensure they represent the expected outcome; ranging from 2 to 12 and being a Gaussian distribution around the value 7. For the final culmination of the problems, the game of Craps, I investigated the rounds where wins and losses took place, found a generalisation of both the chance of winning and length of a game. From this it could be compared to the mathematical equivalents where able and see if it was a good fit to the simulation.

With this information I was able to see if my simulation was accurate and determine that it was in fact a good substitute for a real game of Craps, although I did have to use a specific compiling method: g++ -sdt=c++0x Assignment1.cpp -o Assignment1

2 Part A, Using random Numbers

For part a for my code, that can be seen in the respectively named section at the end of this document, I allowed the input of the number of iterations to be ran. This is so a range of values could be taken to see the trend of the variance as said number of iterations increased through multiple inputs. This was done by a function, rnd(), which ran within a loop for the given number iterations, which generated random doubles between 0 and 1 for each time the loop ran. I then stored each value given in a vector allowing me to calculate the mean and variance. Using this I could, as mentioned, see the trend of the results as the number of iterations increased, and therefore tended to infinity.

For the range of tested values to asses the trend, I decided to use 2^n where n starts at 0, giving 2^0, 1 and ends at 26, giving 2^{26}, 67108864. Looking at the trend allows us to assess the nature of the distribution and as such I compared the variance for the range of values.

Table 1: Sample Size and Variance Results

N	1	2	4	8	16	32	64	128	256
Var(x)	0	0.034934	0.023114	0.460418	0.058175	0.090804	0.073498	0.080076	0.078409

N	512	1024	2048	4096	8192	16384	32768	65536	131072
Var(x)	0.078601	0.079492	0.085488	0.084596	0.08337	0.082908	0.083689	0.083411	0.083181

N	262144	524288	1048576	2097152	4194304	8388608	16777216	33554432	67108864
Var(x)	0.083577	0.083375	0.083286	0.083386	0.083307	0.083334	0.083351	0.083321	0.083341

As can be seen, variance is very volatile at low sample sizes, other than of one, and tend towards the expected mathematical value of $\frac{1}{12}$, $0.0833\dot{3}$, as sample size rapidly increases. Due to this I can say that the following statement is correct:

$$\lim_{x\to\infty} Var(x) = \frac{1}{12} \tag{1}$$

3 Part B, Simulating Dice Throws

For part b of my code, I programmed the simulation of two dice through the use of several functions. I created a dice(N) function where N is the number faces on the dice. Said function works by using the random number generator rnd(), multiplying, then rounding the result. This result is then divided by N and the remainder plus one is equivalent to the value of a N sided dice roll.
This function is used in the function two_dice(N), which finds the sum of dice(N) being ran twice and outputs the integer.

To validate the accuracy and function of the functions the two_dice(N) function is then ran for a six faced dice as two_dice(6) for a range of iterations (100, 500, 1000, 10000, 50000), the results from this are tallied in a vector variable and saved in a respectively named .dat file. This allowed me to then access the data and plot it to see if the results represent the expected Gaussian distribution.
We Know the distribution should be Gaussian due to the probability of rolling each number due to the combinations of possible rolls as seen below:
2={1+1}, 3={2+1, 1+2}, 4={1+3, 3+1, 2+2}, 5={1+4, 4+1, 3+2, 3+2}, 6={1+5, 5+1, 2+4, 4+2, 3+3}, 7={1+6, 6+1, 2+5, 5+2, 3+4, 4+3}, 8={2+6, 6+2, 3+5, 5+3, 4+4}, 9={3+6, 6+3, 4+5, 5+4}, 10={4+6, 6+4, 5+5}, 11={5+6, 6+5}, 12={6+6}
This combination of rolls gives a probability for each roll of the following:
P(2)=$\frac{1}{36}$, P(3)=$\frac{1}{18}$, P(4)=$\frac{3}{36}$, P(5)=$\frac{1}{9}$, P(6)=$\frac{5}{36}$, P(7)=$\frac{1}{6}$, P(8)=$\frac{5}{36}$, P(9)=$\frac{1}{9}$, P(10)=$\frac{3}{36}$, P(11)=$\frac{1}{18}$, P(12)=$\frac{1}{36}$.

As can be seen this is a Gaussian distribution centred around the value 7 due to the possible combination of rolls and if the two_dice(6) function is correct should be seen in its distributions.

Please find the tables and plots of my practical data for the given iterations below.

Table 2: Results of two_dice(6) for a range of iterations

Roll Results	2	3	4	5	6	7	8	9	10	11	12
Results for 100 iterations	5	7	8	11	14	21	12	9	10	1	3
Results for 500 iterations	17	23	49	55	63	73	65	61	49	28	18
Results for 1000 iterations	32	44	84	106	126	168	144	120	104	49	24
Results for 10000 iterations	269	539	858	1121	1383	1600	1404	1135	825	564	303
Results for 50000 iterations	1318	2786	4148	5548	6990	8383	6902	5538	4237	2733	1418

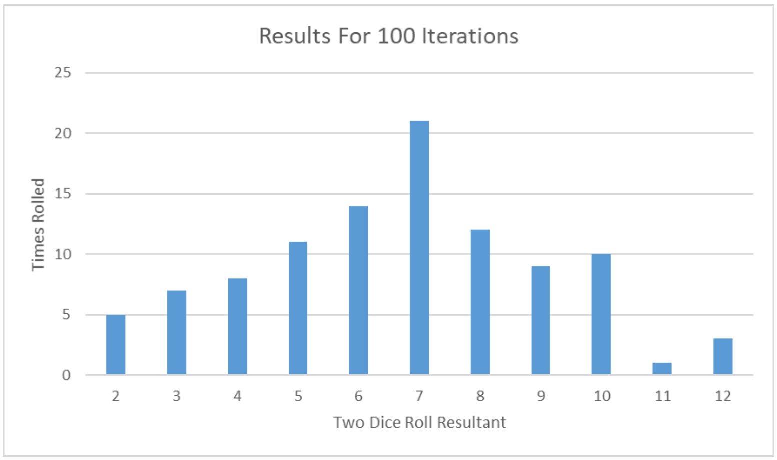

Figure 1: Plot of results for two dice rolled for a given amount of iterations.

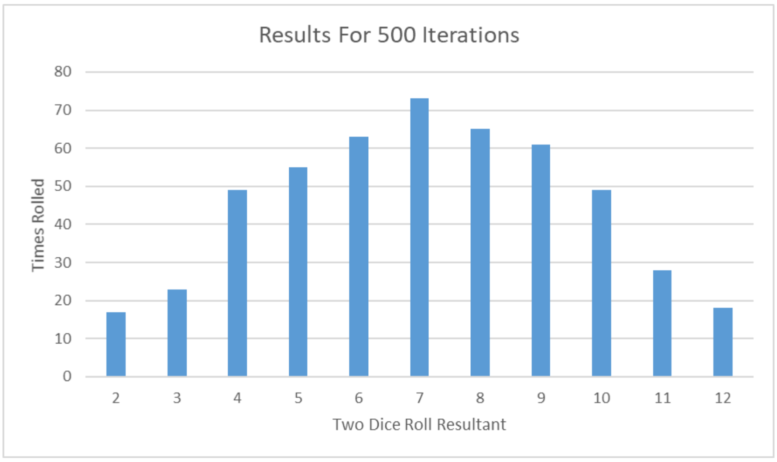

Figure 2: Plot of results for two dice rolled for a given amount of iterations.

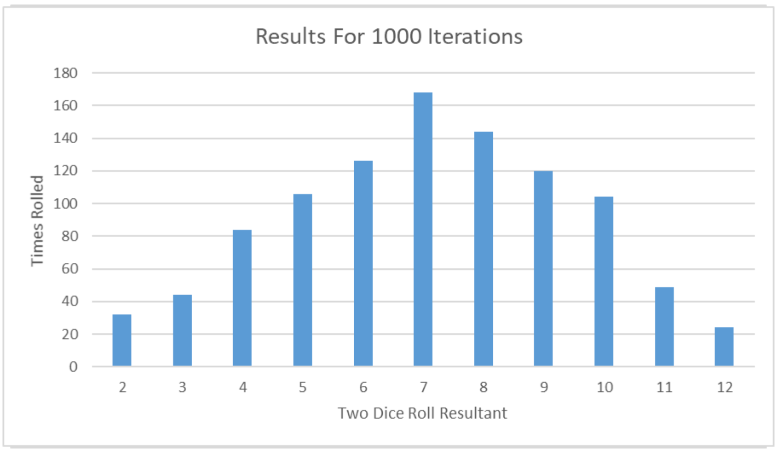

Figure 3: Plot of results for two dice rolled for a given amount of iterations.

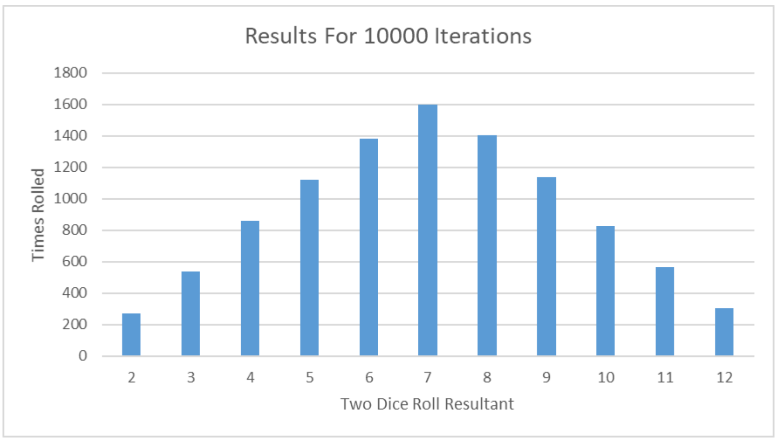

Figure 4: Plot of results for two dice rolled for a given amount of iterations.

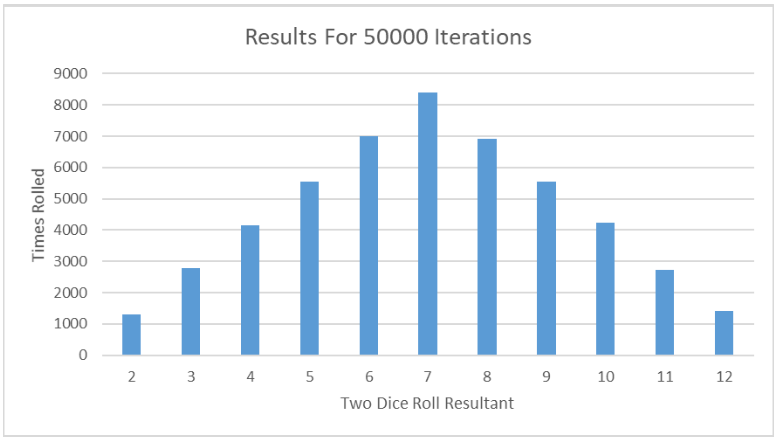

Figure 5: Plot of results for two dice rolled for a given amount of iterations.

As can be seen for the plots of 100 and 500 iterations, figures 1 and 2 respectively, the distribution is more distorted, but still shows resemblance of a Gaussian distribution, this can be seen to increase with sample size. Looking at sample sizes of 10000 and 50000, figures 4 and 5 respectively, for example shows the Gaussian nature of this simulation and therefore its validity, with it meeting the expected distribution as the number of iterations tends to infinity.

4 Part C, The Game of Craps

For part c of my code, I used the two_dice(6) function to simulate a game of Craps. To do so it uses an initial check for the first rounds results where rules differ from the rest of the game, then enters a loop re-rolling until a win or a loss takes place. The game is then replayed until a total of 10000 iterations have been played to give a large sample size to assess; to do so the wins, losses and ratio of wins and losses for each round up to round 20 are all recorded in vectors allowing for me to evaluate the data and compare it to the expected results.

To validate the accuracy of the game I have coded, I plan to look at the mathematical equivalents to the amount of wins and losses seen per round, the chance of winning, when the game should end and when winning the game is most likely. Lets first investigate these probabilities and their ratio's mathematically.

Knowing that a win is a 7 or 11 on round one I can simply add the probability of rolling either to find the chance of winning on the first roll, similarly for losing I add the probabilities of rolling a 2, 3 or 12. Then for round two I know that 2, 3, 7, 11 and 12 weren't rolled previously and can therefore find the probability of rolling the same value twice by adding all the probabilities of the other dice in the roll before and the new roll; therefore the previous roll only has 24 possibilities due to those it couldn't have been (2, 3, 7, 11 and 12) and 36 for the new roll as its undetermined. For losing simply the probability of rolling a 7. For all remaining rolls any value other than 7 could have been rolled and therefore the chance of winning is equivalent to all rolls probabilities of the previous roll and the new roll; similar to the second roll there are now only 30 possibilities due to 7 being the only roll not possible and 36 for the new roll due to being undetermined. For losing, again the chance of this is simply the same as rolling a seven.

$$Win(1) = \frac{1}{6} + \frac{1}{18} = \frac{2}{9} = 0.22222 \tag{2}$$

$$Loss(1) = \frac{1}{36} + \frac{1}{36} + \frac{1}{18} = \frac{1}{9} = 0.11111 \tag{3}$$

$$Ratio(1) = \frac{0.22222}{0.11111} = 2 \tag{4}$$

$$Win(2) = \frac{9 + 16 + 25 + 25 + 16 + 9}{24 * 36} = \frac{100}{864} = 0.11574 \tag{5}$$

$$Loss(2) = \frac{1}{6} = 0.16666 \tag{6}$$

$$Ratio(2) = \frac{0.11574}{0.16666} = 0.69444 \tag{7}$$

$$Win(>2) = \frac{1+4+9+16+25+25+16+9+4+1}{30*36} = \frac{110}{1080} = 0.10185 \tag{8}$$

$$Loss(>2) = \frac{1}{6} = 0.16666 \tag{9}$$

$$Ratio(>2) = \frac{0.10185}{0.16666} = 0.61111 \tag{10}$$

Using this I can see a range of factors: 1) The proportion eliminated per round first is $Win(1) + Loss(1) = 0.33333$, second is $Win(2) + Loss(2) = 0.28241$ and all others $Win(>2) + Loss(>2) = 0.26852$. Therefore I can predict the expected amount of games to end at a given round given a sample size, lets use 10000. Total(0)=10000, Total(1)=(1-0.33333)*10000=6667, Total(2)=(1-0.28241)*6667=4784 and so on, but this shows that over half of games end before the third round which I can compare my code to and use it to determine the wins and losses of each round; note this only indicates the median value of games to have lasted to be the second round not the mean. This and the practical data is plotted below, allowing me to compare I show how many games remain and were won or lost per round as well as the ratio between wins and losses.

Table 3: Shows Both the Practical and Expected Results for a Sample Size of 10000 Games of Craps

	Practical				Expected			
Round	Games Left	Wins	Losses	Ratio	Games Left	Wins	Losses	Ratio
1	6659	2204	1137	1.938434	6666.667	2222.222	1111.111	2
2	4816	772	1071	0.720822	4783.951	771.6049	1111.111	0.694444
3	3527	441	848	0.520047	3499.371	487.2542	797.3251	0.611111
4	2603	340	584	0.582192	2559.725	356.4174	583.2285	0.611111
5	1916	250	437	0.572082	1872.392	260.7128	426.6209	0.611111
6	1417	225	274	0.821168	1369.620	190.7066	312.0653	0.611111
7	1037	139	241	0.576763	1001.852	139.4983	228.2700	0.611111
8	783	101	153	0.660131	732.8358	102.0404	166.9753	0.611111
9	571	86	126	0.68254	536.0558	74.64069	122.1393	0.611111
10	414	68	89	0.764045	392.1149	54.59828	89.34264	0.611111
11	295	49	70	0.7	286.8248	39.93763	65.35249	0.611111
12	215	29	51	0.568627	209.8070	29.21364	47.80413	0.611111
13	161	20	34	0.588235	153.4700	21.36924	34.96784	0.611111
14	118	19	24	0.791667	112.2604	15.63120	25.57833	0.611111
15	86	14	18	0.777778	82.11643	11.43393	18.71007	0.611111
16	65	5	16	0.3125	60.06665	8.363710	13.68607	0.611111
17	46	6	13	0.461538	43.93764	6.117899	10.01111	0.611111
18	35	8	3	2.666667	32.13957	4.475130	7.322940	0.611111
19	21	6	8	0.75	23.50950	3.273475	5.356595	0.611111
20	16	1	4	0.25	17.19676	2.394486	3.918250	0.611111
>20	0	4	12	0.333333				

Looking at this data I can see clear similarities between the expected and practical results. For example the ratio can be of wins to losses can be seen to decrease after round one and two, meaning the chance to win early on is higher than later into the game. Therefore the chance of winning could be said to decrease as the game goes on, as shown by the win to loss ratio.
If I look at the practical data's average ratio of wins and losses for rounds after the second round I get 0.704174 ± 0.125055 for all values included, yet removing anomalous data seen at rounds 18 and 20 I get an average of 0.61545 ± 0.031466; note errors were found using absolute mean error. This is a great representation of that seen in the expected data with its average ratio being 0.611111 giving a similar value to that found experimentally.

With this data, the average round games end is calculated to be 3.446500 ± 0.790665 experimentally and 3.407478 ± 0.747561 for the expected. However the median for both is round two, which would be a better representation due to the skewed nature of the results distribution.
I also calculated the average chance of winning as $47.870 \pm 4.589\%$ for the experimental and $48.102 \pm 4.637\%$ for the expected, note these errors are calculated using mean absolute error and the expected value has an error due to only looking at the first 20 rounds of a game and no further.

5 Conclusion

The game of Craps and it's components all functioned as needed and have proven to do so. The random float used to generate the dice roll is shown to fit it's mathematical expected value due to the variance for uniform distributions being $V(x) = \frac{N}{12}$ as N in this case is simply a range of 1. The two_dice(N) function is shown to be Gaussian in nature and its results export to a .dat file as required for external evaluation in excel or another program able to graphically represent the results. The game itself also showed great similarities to the mathematical expectation for a range of factors. The win to loss ratio was seen to be accurately displayed, both the mean and median round games ended were close, both showed a decrease in chance of winning as the game went on and the average chance of winning was a great fit.

That is why I can say my code to simulate a game of Craps was a great success and as good as the real thing.

6 Code

```
//Compile with: g++ -sdt=c++0x Assignment1.cpp -o Assignment1
//This is to include all wanted functions from modules

#include <iostream>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <typeinfo>
#include <vector>
#include <math.h>
#include <fstream>
#include <sstream>
#include <string>

using namespace std;

double rnd () //function to generate a double with a random value between 1 and 0
{
  float rnd_float; //defines a variable as a float
  rnd_float  = float(rand()) / float( RAND_MAX ); //sets the variables value as that of a
  ↪  random number divided by the highest value of said random number
  return rnd_float; //returns said variable to where the function was called
}

int dice(int n) //function to simulate dice of n sides (n limited to approx 1000 due to
↪  code)
 {
   double rnd_result=rnd (); //sets a variable as a random float between 0 and 1
   int dice_result=(int(round(rnd_result*10000))%n)+1; //uses said variable to generate an
   ↪  integer between 0 and n-1 and adding one
   return dice_result; //returns said generated integer where ther function was called
}

int two_dice(int n) //function to simulate two dice of n sides
 {
   int result; //defines a integer variable
   result=dice(n); //sets value of that of the dice function for a n sided dice
   result+=dice(n); //adds another value of the dice function for an n sided dice
   return result; //returns result variable to where the function was intitially called
}


int main() //main body of the code
{
  srand(time(NULL)); //generates random seed from which to generate random numbers for the
  ↪  whole body of code

  //part A of the code which regards question 1, variables in this segment are notated with
  ↪  an _a to identify them as such
  vector <double> Ans_a; //defines a list of floats to append results to
  double lim_a; //defines a float variable as un upper limit
  cout<< endl<<"Please input random float generator sample size in integer or float
  ↪  format"<< endl; //outputs a prerequisite to the input of the upper limit
  cin >> lim_a; //allows input of the upper limit
  double total_a=0; //sets a variable to total up the random floats generated
  for (int i=0; i<lim_a; i++) //sets up a for loop that repeats for a total amount of times
  ↪  upto the upper limit of generations
  {
    double random_a=rnd(); //uses the rnd() function to generate a float between 0 and 1
    Ans_a.push_back(random_a); //adds the generated float to a list of results
    total_a+=random_a; //adds the float to the total
  }
  double Mean_a=total_a/lim_a; //calculates the mean of the generated results
  double Var_a; //sets up a variable to hold the variance
  //cout<<total_a<<endl<<Mean_a<<endl; //print to see the results from part a code during
  ↪  testing
  for (int i=0; i<lim_a; i++) //an aditional loop that repeats upto the upper limit to
  ↪  access each generated value in the list of results
  {
    Var_a += ((Ans_a.at(i)-Mean_a)*(Ans_a.at(i)-Mean_a)); //sums the (x-mean(x))^2 part of
    ↪  the variance within the loop
  }
  Var_a=Var_a/lim_a; //divides the variance sum by the upper limit to find the true
  ↪  variance
  cout<<"The caclulated variance for a sample size of "<< lim_a << " is " << Var_a << endl;
  ↪  //outputs the results to question 1 /part a of the code
  cout<<"The predicted variance as the sample size increases towards infinity is "<<
  ↪  double(1.0/12.0)<<endl; //outputs the expected results to question 1 /part a of the
  ↪  code

  cout<<endl<<endl; //adds a break in the output

  //part B of the code which regards question 2, variables in this segment are notated with
  ↪  an _b to identify them as such
  vector <int> Ans_b; //defines a list of integers in which we can tally results in
  for(int i=0; i<=10;i++) //loops 11 times to give the amount of possible outcomes of adding
  ↪  two six sided dice
    {
      Ans_b.push_back(0); //sets all tally inputs as the intial value of zero
    }
  vector <int> NV_b; //sets up a vector to hold the wanted iterations to be ran which is N
  ↪  giving NV_b the N vector in part b
  NV_b.push_back(100); //adds a wanted iteration which will be ran (100)
  NV_b.push_back(500); //adds a wanted iteration which will be ran (500)
  NV_b.push_back(1000); //adds a wanted iteration which will be ran (1000)
  NV_b.push_back(10000); //adds a wanted iteration which will be ran (10000)
  NV_b.push_back(50000); //adds a wanted iteration which will be ran (50000)
  //cout<<NV_b.size()<<endl; //used to test .size() function to use in a for loop and see if
  ↪  all values added to the vector during testing
  for(int N=0; N<NV_b.size(); N++) //runs a loop to allow access using Sample_Vector.at(N)
  ↪  by running an integer from 0 to Sample_Vector.size()
    {
      for (int iterations=0; iterations<= NV_b.at(N);iterations++) //runs for the number of
      ↪  iterations as specified in the iteration vector
      {
        int result_b=two_dice(6); //runs the two dice function for a six sided die
        Ans_b.at(result_b-2)+=1; //adds result to the result tally vector

      }
      for(int i=0; i<=10;i++) //loops 11 times to give the amount of possible outcomes of
      ↪  adding two six sided dice
        {
          string Ns=to_string(NV_b.at(N)); //creates a string of the number of iterations
          string filename = "dice_" + Ns  + ".dat"; //sets filename to be used when writing
          ↪  to .DAT file
          //cout<<filename<<endl; //used to diagnose issues with to_string() function, had
          ↪  to compile using g++ -std=c++0x filename.cpp -o filename
          ofstream outfile (filename); //creates or sets file to output to using the defined
          ↪  output filename
          outfile << Ans_b.at(i) <<"\n"; //writes the result tally to the output file
          cout<<Ans_b.at(i)<<" rolls of "<< i+2 << " in a sample of "<<NV_b.at(N)<<endl;
          ↪  //outputs values of all results for each iteration
          Ans_b.at(i)=0; //sets all tally inputs as the intial value of zero
        }
    }

  cout<<endl<<endl; //adds a break in the output

  //part C of the code which regards question 3, variables in this segment are notated with
  ↪  _c to identify them as such
  vector <int>  Wins_c; //defines a vector to hold the rounds where wins take place
  vector <int>  Losses_c; //defines a vector to hold the rounds where losses take place
  vector <double>  WLRatio_c; //defines a vector to hold the ratio of wins to losses in each
  ↪  round
  for(int i=0;i<=20;i++) //loop to set up empty values in each vector to add a tally to
    {
      Wins_c.push_back(0); //sets empty value as stated
      Losses_c.push_back(0); //sets empty value as stated
      WLRatio_c.push_back(0); //sets empty value as stated
    }
  for(int iterations_c=0;iterations_c<10000;iterations_c++) //loops for 10000 iterations as
  ↪  the problem asks
    {
      int check_c=0; //sets up a variable to later allow the breaking of a while loop
      int counter_c=0; //sets or resets a counting variable (upto 20) for placing values in
      ↪  the previously defined vectors
      int result_c=two_dice(6); //runs the two_dice function to define the initial result
      if (result_c==7 or result_c==11) //checks if the initial result is a winning result
        {
          Wins_c.at(0)+=1; //adds a tally to the win vector at position 0 (round 1)
        }
      else if(result_c==2 or result_c==3 or result_c==12) //checks if the initial result is
      ↪  a losing result
        {
          Losses_c.at(0)+=1; //adds a tally to the losses vector at possition 0 (round 1)
        }
      else //if the initial reult was neither a loss or win the following code takes place
        while(check_c==0) //loops while check_c is zero and will break when changed by a win
        ↪  or loss
          {
            if (counter_c<20) //checks check_c is bellow 20 to make sure that the vectors
            ↪  aren't exceeded in length when called
              {
                counter_c+=1; //adds one to the counter variable to correctly add to the
                ↪  vector tally position
              }
            int last_result_c=result_c; //sets up the "point" for this round as the previous
            ↪  rounds result
            result_c=two_dice(6); //runs the two_dice function to define this rounds result
            if(result_c==(last_result_c)) //checks if this round result equals the point
            ↪  (last rounds result) and if so is a win
              {
                Wins_c.at(counter_c)+=1;  //adds 1 to the appropriate win tally using the
                ↪  counter variable to reference position within the vector
                check_c=1; //sets check to 1 and breaks the while loop due to the win
              }
            else if(result_c==7) //checks if this rounds result is equal to 7 which is a
            ↪  loss
              {
                Losses_c.at(counter_c)+=1; //adds 1 to the appropriate loss tally using the
                ↪  counter variable to reference position within the vector
```

```
                check_c=1; //sets check to 1 and breaks the while loop due top the loss
              }
          }
    }
  int Total_Wins_c=0; //sets a variable to total all wins to
  for(int i=0;i<20;i++) //loop to access all points on the vectors
    {
      WLRatio_c.at(i)=float(Wins_c.at(i))/float(Losses_c.at(i)); //defines win to loss ratio
      ↪  for each round
      cout<<"Wins at round "<<i+1<<": "<<Wins_c.at(i)<<", Losses at said round : "<<
      ↪  Losses_c.at(i)<<" giving a ratio of : "<< WLRatio_c.at(i)<< endl; //appropriately
      ↪  outputs results of wins, losses and their ratio for each round
      Total_Wins_c+=Wins_c.at(i); //adds each round to the total variable
    }
  Total_Wins_c+=Wins_c.at(20); //adds all rounds above 20 to total wins
  WLRatio_c.at(20)=float(Wins_c.at(20))/float(Losses_c.at(20)); //finds win loss ratio for
  ↪  rounds above 20
  cout<<"Wins at rounds above 20: "<<Wins_c.at(20)<<", Losses at said rounds : "<<
  ↪  Losses_c.at(20)<<" giving a ratio of : "<< WLRatio_c.at(20)<<endl; //outputs results
  ↪  of all round above 209 for wins, losses and their ratio
  cout<<"Total Wins : "<< Total_Wins_c << " Total Losses : "<<(10000-Total_Wins_c)<< endl;
  ↪  //outputs total wins and total losses

  return 0;
}
```