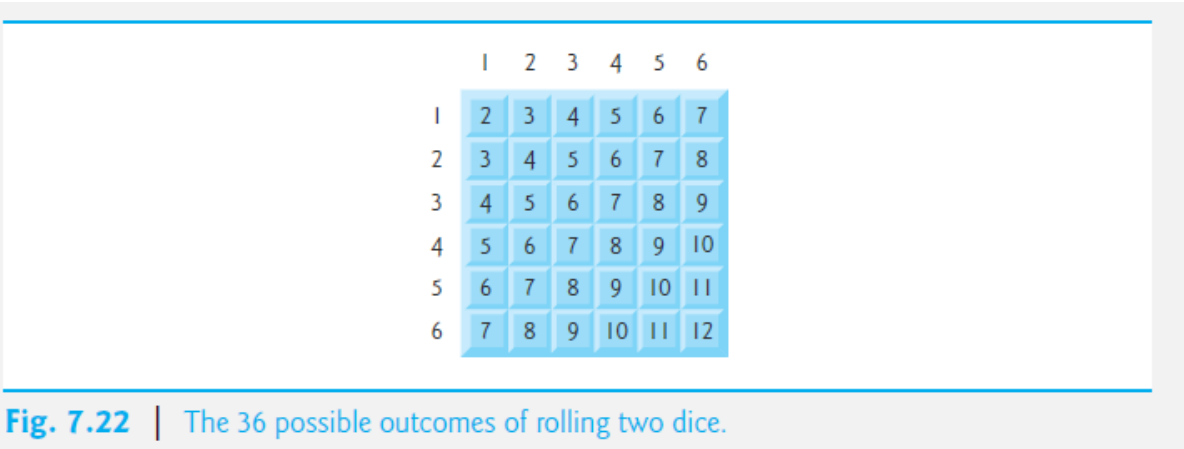
Ngoc Nguyen NITK17K - 1700062

Final Project

Question

(Dice Rolling) Write a program that simulates the rolling of two dice. The sum of the two values



	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 7.22 | The 36 possible outcomes of rolling two dice.

should then be calculated.

[Note: Each die can show an integer value from 1 to 6, so the sum of the two values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 being the least frequent sums.] Figure 7.22 (attached) shows the 36 possible combinations of the two dice. Your program should roll the two dice 36,000,000 times. Use a one-dimensional array to tally the numbers of times each possible sum appears. Print the results in a tabular format. Also, determine if the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one-sixth of all the rolls should be 7).

Github link

https://github.com/NgocNguyenGit/cpp-programming

Internal logic of the program

Objectives

According to the Figure 7.22 attached in the question, it can be analysed that the sum of two dices varies from 2 to 12 with the increment by one. This program is supposed to print the results in a tabular format, in which the first column is the sums (in range from 2 to 12), the second column is the one-dimensional array to tally the numbers of times each possible sum appears (which hereafter is called the 'frequency'), the third column shows the possibilities in theory that the results are expected (in percentage unit, and will be presented later in this document), and the last column shows the actual results solved by the program (in percentage unit).

Expected possibilities calculation

There are 36 possibilities in total to get the sum of two dices and based on the frequency of each sum, the Figure 1.1 shows the possibility of each sum in ratio and in percentage unit. It is the expected result that the program is supposed to achieve.

Sum	Frequency	Possibility (ratio)	Possibility (%)
2	1	1/36	2.778
3	2	2/36	5.556
4	3	3/36	8.333
5	4	4/36	11.111
6	5	5/36	13.889
7	6	6/36 = 1/6	16.667
8	5	5/36	13.889
9	4	4/36	11.111
10	3	3/36	8.333
11	2	2/36	5.556
12	1	1/36	2.778

Figure 1.1. Possibility analysis

Coding phase

In this program, the author would like to apply the class template array instead of the normal array. Therefore, there is a difference in declaring arrays in this program

array<type, arraySize> arrayName;

Based on the Figure 1.1, there are two missing sums which are 0 and 1. Although they are nonsense in this context, the author still counted them in and told the compiler to reserve 13 elements for integer array *frequency*. Accordingly, the array *expected* also included the sums of these two cases which both are 0.

```
//The sum of the two values will vary from 0 to 12
const size_t arraySize{13};

//Initialize all elements to zero
array <unsigned int, arraySize> frequency {};

//Contains counts for the expected number of times
//each sum occurs in 36 rolls of the dice
array <int, arraySize> expected{0, 0, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1};
```

Figure 2.1. Initialisation phase

The array *frequency* was created to count the occurrences of two dices' sum values. The values of all elements in the array *frequency* were assigned to 0. The program should roll the two dices 36,000,000 times, hence, the array frequency must be large enough to store all of the sums' counters and the author used *unsigned int* type for this array.

```
//Use the default random-number generation engine to
//produce uniformly distributed pseudorandom int values from 1 to 6
default_random_engine engine(static_cast<unsigned int>(time(0)));
uniform_int_distribution<unsigned int> randomInt(1,6);
```

Figure 2.2. Random engine

Instead of using simply *srand(time(0))*, the program used the default random-number generation engine to produce uniformly distributed pseudorandom int values from 1 to 6 by including the *<random>* and *<ctime>* libraries.

```
//Roll dice 36,000,000 times
for (unsigned int roll{0}; roll < ROLLS; ++roll) {
    first_dice = randomInt(engine);    //Generates a random number from 1 to 6
    second_dice = randomInt(engine);   //Generates a random number from 1 to 6
    //Increment the appropriate counter in array sum that
    //corresponds to the sum of the first dice and the second dice
    ++frequency[first_dice + second_dice];
}
```

Figure 2.3. For loop rolling two dices 36 million times

As mentioned before, the program used a thirteen-element array in which it ignores the sum values of 0 and 1. Thus, the subscript for array *frequency* in the last line in Figure 2.3 merely varies from 2 to 12. In that line, the calculation's logic was that if e.g., the value of first dice was 1 and so did the second, the counter value of the according sum would increase 1.

```
//Display results of rolling dice
for (size_t j{2}; j < frequency.size(); ++j){
    cout << setw(10) << j << setw(10) << frequency[j] << setprecision(3) << setw(9) <<
        100.0 * expected[j] / 36 << "%" << setprecision(3) << setw(14) << 100.0 *
        frequency[j] / ROLLS << "%"<<endl;
}
```

Figure 2.4. Output the tabular form of results

In order to observe the differences between the expected possibilities and the actual results, the program set the precision of the value up to 3 decimals, and the unit used in the interface was percentage. By comparing the *Theory* and *In program* columns' values, the user can determine if the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one-sixth of all the rolls should be 7). The Figure 2.4 is the loop to output the result.

Text user interface

A screenshot of example program run is enclosed below.

```
       Sum|   Frequency|      Theory|     In program
----------------------------------------------------
         2|      999262|      2.778%|          2.776%
         3|     2002480|      5.556%|          5.562%
         4|     3003476|      8.333%|          8.343%
         5|     4004535|     11.111%|         11.124%
         6|     5000686|     13.889%|         13.891%
         7|     5996800|     16.667%|         16.658%
         8|     4998937|     13.889%|         13.886%
         9|     3996808|     11.111%|         11.102%
        10|     2997458|      8.333%|          8.326%
        11|     1999919|      5.556%|          5.555%
        12|      999639|      2.778%|          2.777%
Program ended with exit code: 0
```

Figure 3.1. Dice-rolling program using array class template

Thanks to the *default_random_engine*, the values of the second and fourth column will be regenerated randomly. Since the errors are small and acceptable, the result from this program is reasonable.

Deficiency of the program

Before applying the class template for array, the author ran a simpler version of this program using normal array and it was observed that the output was printed faster than the current version.

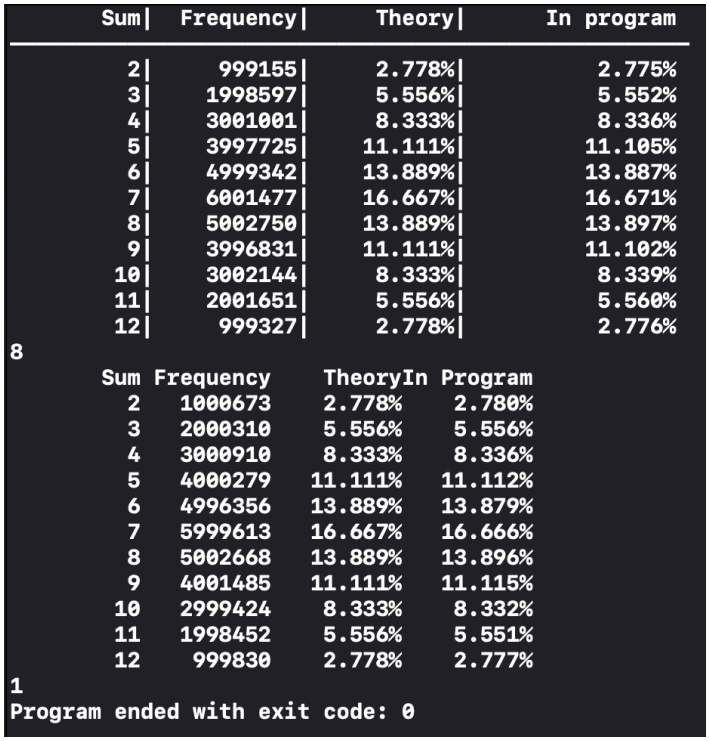
```
const int SIZE = 13;
int frequency_1[SIZE] = {0}; //Array sum.

srand(time(0));          ⚠ Implicit conversion loses integer precision: 'time_t' (aka 'long') to 'unsigned int'

//Roll dice 36,000 times
for (int i = 0; i < ROLLS; i++) {
    first_dice = rand() % 6+1; //Generates a number between 0 and 7
    second_dice = rand() % 6+1; //Generates a number between 0 and 7

    frequency_1[first_dice+second_dice]++; //Find's the sum of the dice roll, and
        increments that's count.
}

cout << setw(10) << "Sum" << setw(10) << "Frequency" << setw(10)
     << "Theory" << setw(10) << "In Program\n" << fixed << showpoint;

//Display results of rolling dice
for (int j = 2; j < SIZE; j++) {
    cout << setw(10) << j << setw(10) << frequency_1[j]
         << setprecision(3) << setw(9)
         << 100.0 * expected[j] / 36 << "%" << setprecision(3)
         << setw(9) << 100.0 * frequency_1[j] / ROLLS << "%\n";
}
```

Figure 4.1. A simpler version of the program (written below the current version, thus, using some initialised variables)

A quick way to calculate the running time of two versions is:

- Adding this line of code at the initialisation phase *time_t tstart, tend;*
- Adding *tstart = time(0);* once before each version
- Adding *tend = time(0);* and *cout<<tend - tstart<<endl;* once after each version

For example, the Figure 4.2 shows that the simpler version only takes 1 second while the class template application version needs 8 seconds to run and show the output.



```
       Sum|  Frequency|     Theory|       In program
__________|___________|___________|________________
         2|     999155|     2.778%|           2.775%
         3|    1998597|     5.556%|           5.552%
         4|    3001001|     8.333%|           8.336%
         5|    3997725|    11.111%|          11.105%
         6|    4999342|    13.889%|          13.887%
         7|    6001477|    16.667%|          16.671%
         8|    5002750|    13.889%|          13.897%
         9|    3996831|    11.111%|          11.102%
        10|    3002144|     8.333%|           8.339%
        11|    2001651|     5.556%|           5.560%
        12|     999327|     2.778%|           2.776%
8
       Sum Frequency    TheoryIn Program
         2   1000673    2.778%    2.780%
         3   2000310    5.556%    5.556%
         4   3000910    8.333%    8.336%
         5   4000279   11.111%   11.112%
         6   4996356   13.889%   13.879%
         7   5999613   16.667%   16.666%
         8   5002668   13.889%   13.896%
         9   4001485   11.111%   11.115%
        10   2999424    8.333%    8.332%
        11   1998452    5.556%    5.551%
        12    999830    2.778%    2.777%
1
Program ended with exit code: 0
```

Figure 4.2. An example to show the running times of two versions

Disclaimer

The program is created in XCode with MacOS; therefore, it will have some unexpected performance if be opened in other operating system or IDE.