CS 201 Homework 02

Bryan Beus

September 16, 2019

Source Code Link: `https://github.com/siddhartha-crypto/cs201/tree/master/hw2`

1 Design

1.1 Main

For the various improvements, I am curious to implement a hashing function, since I have interest in blockchain technology and encryption.

Beyond this aspect, I think the challenge looks relatively straightforward, based on the provided description.

1.2 Money

This challenge is relatively straightforward. I keep the values in integers until the end, to keep the mathematics simple. After all calculations are complete, I then create the dollar value using a float variable and multiplication.

1.3 Rice

For the Rice challenge, I would like to have the current Square we are calculating visually represented, and the number of rice grains next to this square.

Beyond that simple GUI element, I use only standard printouts in the console, to keep the task simple.

2 Post Mortem

2.1 Main

One challenge was deciding between the `size_t` data type and the `string` data type for the hash data. I originally wanted to use `string`, since this is the format for the names. However, the `boost` library that I employed to create the hashes uses `size_t` by default. So, I keep `size_t`, for simplicity's sake.

Another struggle was to match the `" "` space quotation mark in a call to use `std::string(n, c)` to print a number of spaces after each name before printing the hash. I tried using different methods and types, including `size_t` and `char`, but finally realized that simply using single quotation marks around the space within the `std::string()` function was the best solution.

2.2 Money

This challenge was relatively straightforward and did not take a lot of time. I found that the strings floating around were becoming messy, so I placed them all in vectors to keep things simpler, and only passed the vectors.

I did need to use google to remember how to set the floating point precision numbers. `std::fixed` and `std::setprecision(n)` did the trick.

2.3 Rice

This one appeared deceptively simple. What I thought would be a couple of hours of coding expanded into about six. The complexity grew out of the types of variables, and passing them between the functions.

If I had realized it would become so complicated, I wouldn't have spent time on the visual appearance. This only accounted for maybe an hour total of the time, but still, it was unnecessary.

I did gain a fuller appreciation for the build process with C++. The complexity of declaring variables and types in multiple locations forced me to be accurate in a way that JavaScript would not require. I think that C++ was better at helping me avoid mistakes for this reason.

One more note is that the `double` value on my computer never had any issues keeping up with the `unsigned long long int` value. I'm not sure why. Perhaps because my computer is 64 bit?

Also, I had less bugs by declaring the functions first, then writing the code below the `int main()` function. This prevented me from having to check to make sure that a function was declared before it was called.

3 Answers to Questions

- Typical Sizes
  - CHAR: A char is typically an 8-bit (1 byte) value, ideally used for a single character
  - INT: An int can be from positive to negative 2147483647
  - DOUBLE: A double can handle up to 15 digits, so positive or negative 1.79769 e+308
- `#define` is a preprocessor macro. It is used to define a block of text and remains defined until the `#undef` directive is used.
- An initialization is the initial assignment of a value to a variable or data object. An assignment is used at any point in the existance of the variable or data object to change its current value to a potentially different value.
- Technically, when converting a numerical data type from a smaller data type to a type that is capable of holding a wider and larger range of numbers, the conversion can happen

easily and automatically. For example, `char a = 1; int b = 0; b = a;` would not struggle in the processor. However, a developer should seek to define conversions as often as possible, to avoid potential issues.

- A computation is any kind of calculation that can include both arithmetical and non-arithmetical steps.
- An expression is a combination of one or more constants, variables, or operators etc. that combine together to compute another value. A statement is a single line of command, and typically ends with a ; semi-colon.
- A constant expression is a value that cannot change. These are used often in programming to ensure that values that should not change, do not change by fault of the programmer.
- On an int you can perform a bitwise operation (<< shift left).
- With a string, you can store alphanumeric characters.
- This initializes a vector of `char` variables where the initial size of the vector is `20`, and is otherwise empty.

4 Sample Output

Listing 1: "Main"

```
The following hashes belong respectively to the names
   in the names vector.

dorothy:     14174532227680748261
wizard:      15691450355022915147
of:          9028151674594563929
oz:          9549383845444673799
the:         13425634271133782050
man:         6548886666861045578
who:         11765218955868238110
```

```
was:         1938579315280203530
thursday:    17602358137849239207
and:         17573395013354419623

Press enter to continue...
```

Listing 2: Money

```
You have 0 pennies.

You have 1 nickel.

You have 2 dimes.

You have 3 quarters.

The value of all your coins is $1.00

Press enter to continue...
```

Listing 3: Rice

```
                0
       64       9223372036854775808.000000
                9223372036854775808




Int: Total Grains of Rice Collected:
   2147483647
Int: 1000 Grains of Rice Reached on Square:
   10
Int: 1000000 Grains of Rice Reached on Square:
   20
```

```
Int: 1000000000 Grains of Rice Reached on Square:
   30


Double: Total Grains of Rice Collected:
   1.84467e+19
Double: 1000 Grains of Rice Reached on Square:
   10
Double: 1000000 Grains of Rice Reached on Square:
   20
Double: 1000000000 Grains of Rice Reached on Square:
   30


Long: Total Grains of Rice Collected:
   18446744073709551615
The above value is the total of all grains of rice on
   all squares.

The int value tripped on square: 32

Press enter to continue...
```

5 My Programs

5.1 Main

```
/**
 * main.cpp
 * Bryan Beus
 * CS 201
 * September 14, 2019
 * The main program in the assignment - vector names and other features
 */

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

// We use the boost library for hashing names
```

```
15
16 #include <boost/functional/hash.hpp>
17
18 // Store name inputs from the user
19 // The names variable is declared in the main scope
20
21 void InputNames(std::vector<std::string> & names) {
22
23     // Request 10 names from the user using a for loop
24
25     for (int i = 0; i < 10; i++) {
26
27         std::string name;
28         std::cout << "Please enter a name: ";
29         std::getline (std::cin, name);
30
31         // Place each name in the main scope's names vector
32
33         names.push_back(name);
34     }
35 }
36
37 // Wait for the user to indicate that they are ready to continue
38
39 void waitForContinue() {
40
41     std::cout << std::endl << "Press enter to continue...";
42     getchar();
43 }
44
45 // Clear the console
46
47 void clearConsole() {
48
49         // Clear the console
50
51         std::cout << "\033[2J\033[1;1H";
52
53 }
54
55 // Check whether a user-provided name exists within the names data
56
57 bool DoesNameExist(const std::vector<std::string> & names) {
58
59     // Declare and store the user-provided name
60
61     std::string nameToFind;
62     std::cout << "Tell me a name for which to search in the
      ↪  database: ";
63     std::getline (std::cin, nameToFind);
64
65     // Iterate through the names data to see if the name exists
66     for (int i = 0; i < names.size(); i++) {
67         if (names.at(i) == nameToFind) {
68
```

```
69              // If the name does exist, indicate this in the
                ↪  console and return true
70
71              std::cout << "Yes, the name, " << nameToFind << ", is
                ↪  in the data table." << std::endl;
72              return true;
73          }
74      }
75
76      // If the name does not exist, then indicate this in the
        ↪  console and return false
77
78      std::cout << "No, this name is not in the database." <<
        ↪  std::endl;
79      return false;
80  }
81
82  // Find the length of the longest name of the data
83
84  int getLongestNameLength(const std::vector<std::string> & names)
    ↪  {
85
86      // Test that there is at least one name provided in the names
        ↪  data
87
88      if (names.size() < 1) {
89          std::cout << "Warning: There are no names in the data." <<
            ↪  std::endl;
90          return 0;
91      }
92
93      // Declare the variable and initiate it at the first vector
        ↪  value
94
95      int longest_length = names.at(0).length();
96
97      // Iterate through the names data
98      // If any name is longer than the first name, update the
        ↪  variable to the new longest length
99
100     for (int i = 0; i < names.size(); i++) {
101
102         if (longest_length < names.at(i).length()) {
103             longest_length = names.at(i).length();
104         }
105     }
106
107     // Return the longest length
108
109     return longest_length;
110 }
111
112 // Iterate through the names data and print each name to the
    ↪  console
```

```
113
114 void PrintNames(const std::vector<std::string> & names) {
115
116     // Indicate the stage of the function to the user in the
      ↪  console
117
118     std::cout << "The following names are in the database: " <<
      ↪  std::endl << std::endl;
119
120     // Print each name in the names vector
121
122     for (int i = 0; i < names.size(); i++) {
123
124         std::cout << names.at(i) << std::endl;
125     }
126
127 }
128
129 // Create a hash of each name in the names data, place the hash
   ↪  into a table, and return for later use
130
131 std::vector<std::size_t> CreateHashData(const
   ↪  std::vector<std::string> & names) {
132
133     // Declare the hash table
134
135     std::vector<std::size_t> hash_table;
136
137     // For each name in names, use the boost library to create a
      ↪  hash and place it into the table
138
139     for (int i = 0; i < names.size(); i++) {
140         boost::hash<std::string> string_hash;
141         std::size_t hashed_name = string_hash(names.at(i));
142         hash_table.push_back(hashed_name);
143     }
144
145     // Return the hash_table variable
146
147     return hash_table;
148
149 }
150
151 // Print the names and the associated name hashes in the data
152
153 void PrintNameHashes(const std::vector<std::size_t> & hash_table,
   ↪  const std::vector<std::string> & names) {
154
155     // State in the console what the function does
156
157     std::cout << "The following hashes belong respectively to the
      ↪  names in the names vector." << std::endl << std::endl;
158
```

```
159     // Declare a longest_length varaible and use the
        ↪  getLongestNameLength() function to create a universal
        ↪  target length of spaces " "
160
161     int longest_length = getLongestNameLength(names);
162
163     // Iterate through the names and hashes and print them to the
        ↪  table
164     for (int i = 0; i < hash_table.size(); i++) {
165
166         // Declare the target number of spaces, given the format
            ↪  for the console output
167         // Add 3 units of spaces onto the variable, for good
            ↪  measure
168
169         int num_spaces = longest_length + 3 -
            ↪  names.at(i).length();
170
171         // Print the names, num_spaces spaces between, and the
            ↪  hashes
172
173         std::cout << names.at(i) << ":" << std::string(num_spaces,
            ↪  ' ') << hash_table.at(i) << std::endl;
174     }
175
176 }
177
178 int main(int argc, char **argv) {
179
180     // Clear the console
181
182     clearConsole();
183
184     // Declare a names variable to serve throughout the program
185
186     std::vector<std::string> names;
187
188     // Call the InputNames() function to request the user to
        ↪  provide names
189
190     InputNames(names);
191
192     // Wait for user permission to continue
193
194     waitForContinue();
195
196     // Clear the console
197
198     clearConsole();
199
200     // Call the DoesNameExist() function to request the user to
        ↪  search for a name
201
202     DoesNameExist(names);
```

```
203
204     // Wait for user permission to continue
205
206     waitForContinue();
207
208     // Clear the console
209
210     clearConsole();
211
212     // Call the PrintNames() function to print names to the
        ↪  console
213
214     PrintNames(names);
215
216     // Create a vector variable with hashes of the names using
        ↪  the CreateHashData() function
217
218     std::vector<std::size_t> hash_table = CreateHashData(names);
219
220     // Wait for user permission to continue
221
222     waitForContinue();
223
224     // Clear the console
225
226     clearConsole();
227
228     // Use the PrintNameHashes() function to print all names and
        ↪  hashes to the console
229     PrintNameHashes(hash_table, names);
230
231     // Wait for user permission to continue
232
233     waitForContinue();
234
235     // Clear the console
236
237     clearConsole();
238
239     // End
240
241     return 0;
242
243 }
```

5.2 Money

```
1 /**
2  * money.cpp
3  * CS 201
4  * Bryan Beus
5  * September 14, 2019
```

```
 * A program to count the money a user has and to return a clean summation of the value
 */

#include <iostream>
#include <string>
#include <vector>
#include <iomanip>

// Clear the console

void clearConsole() {

    // Clear the console

    std::cout << "\033[2J\033[1;1H";

}

// Wait for the user to indicate that they are ready to continue

void waitForContinue() {

    std::cout << std::endl << "Press enter to continue...";
    getchar();
}

// Inform the user their input is invalid

void askUserAgain() {

    std::cout << "You provided an invalid input. Please try again." << std::endl << std::endl;;

}

// Query the user to input their wallet state

void queryUserWallet(std::vector<int> & user_wallet, std::vector<std::string> & coin_list_plural) {

    // Declare an input variable for user input

    int input;

    // Request the user to input the total number of each coin they have in their wallet

    for (int i = 0; i < coin_list_plural.size(); i++) {

        clearConsole();

        std::cout << "How many " << coin_list_plural.at(i) << " do you have? ";

```

```
56          // Initiate a while loop to wait until the user inputs a
            ↪  viable response
57
58          while (true) {
59
60              std::cin >> input;
61
62              // If the response is invalid, ask again
63
64              if (std::cin.fail() || input < 0) {
65                  std::cin.clear();
66                  std::cin.ignore(1000, '\n');
67                  askUserAgain();
68                  waitForContinue();
69
70                  // If the response is valid, input the value and
                    ↪  move to the next iteration of the for loop
71
72              } else {
73                 user_wallet.push_back(input);
74                 std::cin.clear();
75                 std::cin.ignore(1000, '\n');
76                 break;
77              }
78          }
79      }
80  }
81
82  // Calculate the wallet total as a floating point variable
83
84  float calculateWalletTotal(std::vector<int> & user_wallet) {
85
86      float total_wallet = 0;
87
88      // Initiate the various values of the coins
89
90      std::vector<int> values;
91          values.push_back(1);
92          values.push_back(5);
93          values.push_back(10);
94          values.push_back(25);
95
96          // Calcuate the total value of the wallet in pennies
97
98      for (int i = 0; i < 4; i++) {
99          total_wallet = total_wallet + user_wallet.at(i) *
            ↪  values.at(i);
100     }
101
102     // Transform the total value into a dollar value
103
104     total_wallet = total_wallet * 0.01;
105
106     // Return the total value
```

```
107
108     return total_wallet;
109 }
110
111 // Print to the console the total wallet sum
112
113 void reportWalletSum(std::vector<int> & user_wallet,
    ↪  std::vector<std::string> & coin_list_plural,
    ↪  std::vector<std::string> & coin_list_singular) {
114
115     clearConsole();
116
117     // Call the calculateWalletTotal() function to calculate the
        ↪  wallet total
118
119     float total_wallet = calculateWalletTotal(user_wallet);
120
121     // For each coin type, print the total in the user's wallet
122
123     for (int i = 0; i < 4; i++) {
124
125         std::cout << "You have " << user_wallet.at(i) << " ";
126
127         if (user_wallet.at(i) == 1) {
128             std::cout << coin_list_singular.at(i);
129         } else {
130             std::cout << coin_list_plural.at(i);
131         }
132
133         std::cout << "." << std::endl << std::endl;
134
135     }
136
137     // Print the total value in the wallet
138
139     std::cout << "The value of all your coins is $" << std::fixed
        ↪  << std::setprecision(2) << total_wallet << std::endl;
140
141     // Pause for user to continue
142
143     waitForContinue();
144
145 }
146
147 int main(int argc, char **argv) {
148
149     // Clear the console
150
151     clearConsole();
152
153     // Declare the vector to hold the user's coin totals
154
155     std::vector<int> user_wallet;
156
157     // Create list of plural coin names
```

```
158
159     std::vector<std::string> coin_list_plural;
160         coin_list_plural.push_back("pennies");
161         coin_list_plural.push_back("nickels");
162         coin_list_plural.push_back("dimes");
163         coin_list_plural.push_back("quarters");
164
165     // Create list of singular coin names
166
167     std::vector<std::string> coin_list_singular;
168         coin_list_singular.push_back("penny");
169         coin_list_singular.push_back("nickel");
170         coin_list_singular.push_back("dime");
171         coin_list_singular.push_back("quarter");
172
173         // Query the user's wallet
174
175     queryUserWallet(user_wallet, coin_list_plural);
176
177     // Clear the console
178
179     clearConsole();
180
181     // Report the total value
182
183     reportWalletSum(user_wallet, coin_list_plural,
        ↪  coin_list_singular);
184
185     return 0;
186 }
```

5.3 Rice

```
1  /**
2   * rice.cpp
3   * CS 201
4   * Bryan Beus
5   * September 15, 2019
6   * A program to display the power of compound interest and to
    ↪  observe the output in various variable types
7   */
8
9  #include <iostream>
10 #include <string>
11 #include <vector>
12
13 // Define the Width of the blank spaces in the cells
14 // This must be an even number
15
16 #define Width 6
17
```

```
18 // Clear the console
19
20 void clearConsole();
21
22 // Wait for the user to indicate that they are ready to continue
23
24 void waitForContinue();
25
26 // Set a default function to print a series of blank spaces of
   ↪  length <Width>
27
28 void print_full_width(int longest_length, int col_type);
29
30 // Set a default function to print a series of blank spaces of
   ↪  half of length <Width>
31
32 void print_half_width();
33
34 // Set a default function to print a series of double bars of
   ↪  length <Width>
35
36 void print_full_bar(int longest_length, int col_type);
37
38 // Print the top of the grid
39
40 void print_top_line(int longest_length);
41
42 // Fill a whole row that has no variables or grid corners
43
44 void print_fill_row(int longest_length);
45
46 // Fill a row that has variables, including row numbers and
   ↪  variables inside the grid boxes
47 // Row requires both the current row to print and a vector that
   ↪  has the current state of grid boxes (X's or .'s)
48
49 void printSquare(int & currentSquare);
50
51 // Print a row that will have at least one variable value on it.
52
53 void print_var_row(int & currentSquare, int & longest_length,
   ↪  std::vector<std::string> & current_total_string);
54
55 // Print the bottom line of the grid
56
57 void print_bottom_line(int & longest_length);
58
59 // Print the current square
60
61 void printCurrentSquare(int & currentSquare,
   ↪  std::vector<std::string> & current_total_string);
62
63 // Calculate new values for each important value
64
```

```cpp
void calculateNewValues(int & total_in_int, double & total_in_double, unsigned long long int & total_in_long, int & full_total_in_int, double & full_total_in_double, unsigned long long int & full_total_in_long, int & square_int_tripped, int & square_double_tripped);

// Create a vector of strings that represent the current state of the variables
// This helps in formatting the GUI table

void createTotalString(std::vector<std::string> & current_total_string, int & total_in_int, double & total_in_double, unsigned long long int & total_in_long);

// Print the measurements for the challenge questions

void printMeasurements(int & currentSquare, int & full_total_in_int, double & full_total_in_double, unsigned long long int & full_total_in_long, std::vector<int> & values_met_int,  std::vector<double> & values_met_double, int & square_int_tripped, int & square_double_tripped, int & total_in_int, double & total_in_double);

int main(int argc, char **argv) {

    // Declare the variables that represent the grains of rice on a single square for the currently calculated square

    int total_in_int = 1;
    double total_in_double = 1;
    unsigned long long int total_in_long = 1;

    // Declare a vector to hold the string representation of the digital values
    // This is useful for formatting purposes

    std::vector<std::string> current_total_string;

    // Declare variables to represent the sum total of all grains of rice collected

    int full_total_in_int = total_in_int;
    double full_total_in_double = total_in_double;
    unsigned long long int full_total_in_long = total_in_long;

    // Declare vectors to track the square numbers at which our challenge questions are met

    std::vector<int> values_met_int;
    for (int i = 0; i < 3; i++) {
        values_met_int.push_back(0);
    }

```

```
    std::vector<double> values_met_double;
    for (int i = 0; i < 3; i++) {
        values_met_double.push_back(0);
    }

    // Declare variables to check when a value type might fail to keep up with the total numbers

    int square_int_tripped = 0;
    int square_double_tripped = 0;

    // Declare variables to track the current square and total

    int currentSquare = 1;
    int totalSquares = 64;

    // Clear the console before we begin

    clearConsole();

    // Initiate a while loop for all calculations and displays

    while (currentSquare <= totalSquares) {

        // Call the createTotalString function to create the string representations of our grains of rice on the current square

        createTotalString(current_total_string, total_in_int, total_in_double, total_in_long);

        // Display the current square

        printCurrentSquare(currentSquare, current_total_string);

        // Print the measurements that track our challenge questions

        printMeasurements(currentSquare, full_total_in_int, full_total_in_double, full_total_in_long, values_met_int, values_met_double, square_int_tripped, square_double_tripped, total_in_int, total_in_double);

       // Wait for the user to indicate they are ready to proceed to the next square

        waitForContinue();

        // Clear the console before proceeding
        clearConsole();

        // Calculate the values for the next square

```

```
146         calculateNewValues(total_in_int, total_in_double,
                ↪  total_in_long, full_total_in_int,
                ↪  full_total_in_double, full_total_in_long,
                ↪  square_int_tripped, square_double_tripped);
147
148         // Increase our total square count
149
150         ++currentSquare;
151     }
152
153     return 0;
154 }
155
156 // Clear the console
157
158 void clearConsole() {
159
160     std::cout << "\033[2J\033[1;1H";
161
162 }
163
164 // Wait for the user to indicate that they are ready to continue
165
166 void waitForContinue() {
167
168     std::cout << std::endl << "Press enter to continue...";
169     getchar();
170 }
171
172 // Set a default function to print a series of blank spaces of
    ↪  length <Width>
173
174 void print_full_width(int longest_length, int col_type) {
175
176     // If the column is on the left, print a Width-wide row of
        ↪  blank spaces
177
178     if (col_type == 0) {
179         for (int i = 0; i < Width; i++) {
180             std::cout << " ";
181         }
182
183     // If the column is on the right, print a row of blank spaces
        ↪  that appropriately matches the length of the longest
        ↪  number of grains of rice
184
185     } else if (col_type == 1)
186         for (int i = 0; i < longest_length + (Width * 2 / 3); i++)
            ↪  {
187             std::cout << " ";
188     }
189 }
190
```

```
191 // Set a default function to print a series of blank spaces of
    ↪  half of length <Width>
192
193 void print_half_width() {
194     for (int j = 0; j < (Width * 1 / 3); j++) {
195         std::cout << " ";
196     }
197 }
198
199 // Set a default function to print a series of double bars of
    ↪  length <Width>
200
201 void print_full_bar(int longest_length, int col_type) {
202
203     // If the column is on the left, print a bar of Width length
204
205     if (col_type == 0) {
206         for (int i = 0; i < Width; i++) {
207             std::cout << "";
208         }
209
210     // If the column is on the right, print a bar of a length
        ↪  appropriate for the longest number of grains of rice
211
212     } else if (col_type == 1)
213         for (int i = 0; i < longest_length + (Width * 2 / 3); i++)
            ↪  {
214             std::cout << "";
215     }
216 }
217
218 // Print the top of the grid
219
220 void print_top_line(int longest_length) {
221
222     // Vertically clear at least one line in the terminal, then
        ↪  print the <Width> blank spaces
223
224     std::cout << std::endl;
225
226     // Print the top row of the grid
227
228     std::cout << "  ";
229
230     print_full_bar(longest_length, 0);
231
232     std::cout << "";
233
234     print_full_bar(longest_length, 1);
235
236     std::cout << "" << std::endl;
237 }
238
239 // Fill a whole row that has no variables or grid corners
```

```
240
241 void print_fill_row(int longest_length) {
242
243     // Print a divider bar with some formatting spaces
244
245     std::cout << "  ";
246
247     // Call the print_full_width() function to print the left
      ↪  column
248
249     print_full_width(longest_length, 0);
250
251     // Print a divider bar
252
253     std::cout << "";
254
255     // Call the print_full_width() function to print the right
      ↪  column
256
257     print_full_width(longest_length, 1);
258
259     // Print a divider bar
260
261     std::cout << "" << std::endl;
262
263 }
264
265 // Fill a row that has variables, including row numbers and
   ↪  variables inside the grid boxes
266
267 void printSquare(int & currentSquare) {
268
269     // Call default function to print half width of spaces
270
271     print_half_width();
272
273     // If the current square number is less than 10, add an extra
      ↪  space for formatting
274
275     if (currentSquare < 10) {
276         std::cout << " ";
277     }
278
279     // Print the current square number
280
281     std::cout << currentSquare;
282
283     // Call default function to print half width of spaces
284
285     print_half_width();
286
287
288 }
289
```

```
290 // Print a row in the rice/square GUI element that has variables
    ↪  on it
291
292 void print_var_row(int & currentSquare, int & longest_length,
    ↪  std::vector<std::string> & current_total_string) {
293
294     // Iterate through each of the rows
295
296     for (int i = 0; i < 3; i++) {
297
298         // Print the first divider bar
299
300         std::cout << "  ";
301
302         // If this is the second row, print the square number
303
304         if (i == 1) {
305             printSquare(currentSquare);
306
307         // Otherwise, keep the first column blank
308
309         } else {
310             print_full_width(longest_length, 0);
311         }
312
313         // Divider bar
314
315         std::cout << "";
316
317         // Print a bit of extra space for formatting, before
            ↪  printing rice grain numbers
318
319         print_half_width();
320
321         // Check how many blank spaces are needed to keep the
            ↪  current number in sync with the format of the grid
322
323         int num_spaces = longest_length -
            ↪  current_total_string.at(i).length();
324
325         // Print the number of grains of rice, and the necessary
            ↪  blank spaces for formatting
326         std::cout << current_total_string.at(i) <<
            ↪  std::string(num_spaces, ' ');
327
328         // Print some more padding
329
330         print_half_width();
331
332         // Final divider bar
333
334         std::cout << "" << std::endl;
335     }
336
```

```
}

// Print the bottom line of the grid

void print_bottom_line(int & longest_length) {

    // Print bottom corner

    std::cout << "  ";

    // Print a full bar of appropriate length for left column

    print_full_bar(longest_length, 0);

    // Print divider

    std::cout << "";

    // Print a full bar of appropriate length for right column

    print_full_bar(longest_length, 1);

    // Print right bottom corner

    std::cout << "" << std::endl;
}

// Print the current square number, with the appropriate number of empty spaces around it

void printCurrentSquare(int & currentSquare, std::vector<std::string> & current_total_string) {

    // Calculate the longest length of the three records

    int longest_length = current_total_string.at(0).length();

    for (int i = 1; i < current_total_string.size(); i++) {

        if (longest_length < current_total_string.at(i).length()) {
            longest_length = current_total_string.at(i).length();
        }
    }

    // Print first rows of grid

    print_top_line(longest_length);
    print_fill_row(longest_length);

    // Print the variable rows

    print_var_row(currentSquare, longest_length, current_total_string);

```

```
    // the bottom rows of grid

    print_fill_row(longest_length);
    print_bottom_line(longest_length);

}

// Calculate new values for each of the important variables; Call this function after printing the current variables to the console

void calculateNewValues(int & total_in_int, double & total_in_double, unsigned long long int & total_in_long, int & full_total_in_int, double & full_total_in_double, unsigned long long int & full_total_in_long, int & square_int_tripped, int & square_double_tripped) {

    // Double the current values of grains of rice on the square

    total_in_int = 2 * total_in_int;
    total_in_double = 2 * total_in_double;
    total_in_long = 2 * total_in_long;

    // While ensuring that we're not adding negatives or zeros (should the size increase beyond capacity), add the current square's rice to the running total for each variable type

    if (total_in_int >= 1) {
        full_total_in_int = full_total_in_int + total_in_int;
    }

    if (total_in_double >= 1) {
        full_total_in_double = full_total_in_double + total_in_double;
    }

    if (total_in_long >= 1) {
        full_total_in_long = full_total_in_long + total_in_long;
    }

}

// Create a string that can visually represent the state of the current square's grains of rice count
// This is useful for formatting

void createTotalString(std::vector<std::string> & current_total_string, int & total_in_int, double & total_in_double, unsigned long long int & total_in_long) {

    // Clear the current_total_string vector

```

```
428     current_total_string.clear();
429 
430     // Add in the new numbers as strings
431 
432     current_total_string.push_back(std::to_string(total_in_int));
433 
        ↪  current_total_string.push_back(std::to_string(total_in_double));
434    current_total_string.push_back(std::to_string(total_in_long));
435 
436 }
437 
438 // Print the current measurements that answer the challenge
    ↪  questions
439 
440 void printMeasurements(int & currentSquare, int &
    full_total_in_int, double & full_total_in_double, unsigned
    ↪  long long int & full_total_in_long, std::vector<int> &
    ↪  values_met_int,  std::vector<double> & values_met_double, int
    ↪  & square_int_tripped, int & square_double_tripped, int &
    ↪  total_in_int, double & total_in_double) {
441 
442     // For each data type and for each of the three standards
          ↪  that we want to measure in the challenge questions, check
          ↪  to see whether or not we have surpassed that number of
          ↪  grains of rice
443     // If we have, add this value to our vector that tracks the
          ↪  square on which this event occurs
444 
445     if (full_total_in_int >= 1000 && values_met_int.at(0) == 0) {
446 
447         values_met_int.at(0) = currentSquare;
448     }
449 
450     if (full_total_in_int >= 1000000 && values_met_int.at(1) ==
          ↪  0) {
451         values_met_int.at(1) = currentSquare;
452     }
453 
454     if (full_total_in_int >= 1000000000 && values_met_int.at(2)
          ↪  == 0) {
455         values_met_int.at(2) = currentSquare;
456     }
457 
458     if (full_total_in_double >= 1000 && values_met_double.at(0)
          ↪  == 0) {
459 
460         values_met_double.at(0) = currentSquare;
461     }
462 
463     if (full_total_in_double >= 1000000 &&
          ↪  values_met_double.at(1) == 0) {
464         values_met_double.at(1) = currentSquare;
465     }
```

```
466
467     if (full_total_in_double >= 1000000000 &&
        ↪  values_met_double.at(2) == 0) {
468         values_met_double.at(2) = currentSquare;
469     }
470
471     // Add an extra space for formatting
472
473     std::cout << std::endl;
474
475     // Print the running total of grains of rice, according to
        ↪  the int data type
476
477         std::cout << "Int: Total Grains of Rice Collected:
            ↪  " << full_total_in_int << std::endl;
478
479     // For the int data type, for each of the three standards we
        ↪  measure, when they occur print them to the console
480
481     if (values_met_int.at(0) > 0) {
482         std::cout << "Int: 1000 Grains of Rice Reached on Square:
            ↪  " << values_met_int.at(0) << std::endl;
483     }
484
485     if (values_met_int.at(1) > 0) {
486         std::cout << "Int: 1000000 Grains of Rice Reached on
            ↪  Square:       " << values_met_int.at(1) << std::endl;
487     }
488
489     if (values_met_int.at(2) > 0) {
490         std::cout << "Int: 1000000000 Grains of Rice Reached on
            ↪  Square:    " << values_met_int.at(2) << std::endl <<
            ↪  std::endl;
491     }
492
493     // Add an extra space for formatting
494
495     std::cout << std::endl;
496
497     // Print the running total of grains of rice, according to
        ↪  the double data type
498
499         std::cout << "Double: Total Grains of Rice Collected:
            ↪  " << full_total_in_double << std::endl;
500
501     // For the double data type, for each of the three standards
        ↪  we measure, when they occur print them to the console
502
503     if (values_met_double.at(0) > 0) {
504         std::cout << "Double: 1000 Grains of Rice Reached on
            ↪  Square:       " << values_met_double.at(0) <<
            ↪  std::endl;
505     }
```

```
506
507     if (values_met_double.at(1) > 0) {
508         std::cout << "Double: 1000000 Grains of Rice Reached on
            ↪  Square:    " << values_met_double.at(1) << std::endl;
509     }
510
511     if (values_met_double.at(2) > 0) {
512         std::cout << "Double: 1000000000 Grains of Rice Reached
            ↪  on Square: " << values_met_double.at(2) << std::endl
            ↪  << std::endl;
513     }
514
515     // Add an extra space for formatting
516
517     std::cout << std::endl;
518
519     // Print the running total in the long data type
520
521         std::cout << "Long: Total Grains of Rice Collected:
            ↪  " << full_total_in_long << std::endl;
522
523     // When we reach the end of all calculations, print our
        ↪  result in the console
524
525     if (currentSquare == 64) {
526         std::cout << "The above value is the total of all grains
            ↪  of rice on all squares." << std::endl;
527     }
528
529     // Add an extra space for formatting
530
531     std::cout << std::endl;
532
533     // Calculate the square on which the int or the double data
        ↪  type may stop keeping up with our running total
534
535     if (square_int_tripped == 0 && total_in_int <= 0) {
536         square_int_tripped = currentSquare;
537     }
538
539     if (square_double_tripped == 0 && total_in_double <= 0) {
540         square_double_tripped = currentSquare;
541     }
542
543     // Report the square on which any failed data type
        ↪  experienced the failure
544
545     if (square_int_tripped != 0) {
546
547         std::cout << "The int value tripped on square: " <<
            ↪  square_int_tripped << std::endl;
548     }
549
```

```
    if (square_double_tripped != 0) {

        std::cout << "The double value tripped on square: " <<
        ↪  square_int_tripped;
    }

}
```