CS 201 Homework 02

Bryan Beus

September 23, 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.

1.4 Scores

I do not intend to use any design elements for this one, but rather will focus on making sure that I prevent as many bugs as possible.

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.

2.4 Scores

One challenge in the design of this module was to ensure that each vector scaled according to the other.

Sometimes, if I put a name into the names vector, and then fed an improper input (such as a character) to the scores integer, my program design would still record and expand the names vector while preventing the scores vector from expanding.

I added a failsafe that ends the program, in the event that either vector scales without the other matching, and experimented with many potential error paths until I was fairly certain the vectors will scale properly.

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:   17603258137849239207
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...
```

Listing 4: Scores

```
Please enter names and scores for the database.

To indicate that you are finished entering data, input
    a name as "NoName" paired with a score of "0"

Enter a new name for the database: Rocky
Enter the score for Rocky: 151
Enter a new name for the database: Dustin
Enter the score for Dustin: 522
Enter a new name for the database: Trevor
Enter the score for Trevor: 322
Enter a new name for the database: Parker
Enter the score for Parker: 151
Enter a new name for the database: NoName

Please confirm by entering '0' as a score.
```

5 My Programs

5.1 Main

```cpp
/**
 * 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

#include <boost/functional/hash.hpp>

// Store name inputs from the user
// The names variable is declared in the main scope

void InputNames(std::vector<std::string> & names) {

    // Request 10 names from the user using a for loop

    for (int i = 0; i < 10; i++) {

        std::string name;
        std::cout << "Please enter a name: ";
        std::getline (std::cin, name);

        // Place each name in the main scope's names vector

        names.push_back(name);
    }
}

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

void waitForContinue() {

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

// Clear the console

void clearConsole() {

        // Clear the console

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

}

// Check whether a user-provided name exists within the names data

bool DoesNameExist(const std::vector<std::string> & names) {

    // Declare and store the user-provided name

    std::string nameToFind;
    std::cout << "Tell me a name for which to search in the
     ↪  database: ";
    std::getline (std::cin, nameToFind);

    // Iterate through the names data to see if the name exists
    for (int i = 0; i < names.size(); i++) {
        if (names.at(i) == nameToFind) {
            // If the name does exist, indicate this in the
             ↪  console and return true

            std::cout << "Yes, the name, " << nameToFind << ", is
             ↪  in the data table." << std::endl;
            return true;
        }
    }

    // If the name does not exist, then indicate this in the
     ↪  console and return false

    std::cout << "No, this name is not in the database." <<
     ↪  std::endl;
    return false;
}

// Find the length of the longest name of the data

int getLongestNameLength(const std::vector<std::string> & names)
 ↪  {

    // Test that there is at least one name provided in the names
     ↪  data

    if (names.size() < 1) {
        std::cout << "Warning: There are no names in the data." <<
         ↪  std::endl;
        return 0;
    }

    // Declare the variable and initiate it at the first vector
     ↪  value

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

    // Iterate through the names data
    // If any name is longer than the first name, update the
     ↪  variable to the new longest length

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

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

    // Return the longest length

    return longest_length;
}

```

```cpp
// Iterate through the names data and print each name to the
↪  console

void PrintNames(const std::vector<std::string> & names) {

    // Indicate the stage of the function to the user in the
    ↪  console

    std::cout << "The following names are in the database: " <<
    ↪  std::endl << std::endl;

    // Print each name in the names vector

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

        std::cout << names.at(i) << std::endl;
    }

}

// Create a hash of each name in the names data, place the hash
↪  into a table, and return for later use

std::vector<std::size_t> CreateHashData(const
↪  std::vector<std::string> & names) {

    // Declare the hash table

    std::vector<std::size_t> hash_table;

    // For each name in names, use the boost library to create a
    ↪  hash and place it into the table

    for (int i = 0; i < names.size(); i++) {
        boost::hash<std::string> string_hash;
        std::size_t hashed_name = string_hash(names.at(i));
        hash_table.push_back(hashed_name);
    }

    // Return the hash_table variable

    return hash_table;

}

// Print the names and the associated name hashes in the data

void PrintNameHashes(const std::vector<std::size_t> & hash_table,
↪  const std::vector<std::string> & names) {

    // State in the console what the function does

    std::cout << "The following hashes belong respectively to the
    ↪  names in the names vector." << std::endl << std::endl;

    // Declare a longest_length varaible and use the
     ↪  getLongestNameLength() function to create a universal
     ↪  target length of spaces " "

    int longest_length = getLongestNameLength(names);

    // Iterate through the names and hashes and print them to the
     ↪  table
    for (int i = 0; i < hash_table.size(); i++) {

        // Declare the target number of spaces, given the format
         ↪  for the console output
        // Add 3 units of spaces onto the variable, for good
         ↪  measure

        int num_spaces = longest_length + 3 -
         ↪  names.at(i).length();

        // Print the names, num_spaces spaces between, and the
         ↪  hashes

       std::cout << names.at(i) << ":" << std::string(num_spaces,
         ↪  ' ') << hash_table.at(i) << std::endl;
    }

}

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

    // Clear the console

    clearConsole();

    // Declare a names variable to serve throughout the program

    std::vector<std::string> names;

    // Call the InputNames() function to request the user to
     ↪  provide names

    InputNames(names);

    // Wait for user permission to continue

    waitForContinue();

    // Clear the console

    clearConsole();

    // Call the DoesNameExist() function to request the user to
     ↪  search for a name

    DoesNameExist(names);

    // Wait for user permission to continue

    waitForContinue();

    // Clear the console

    clearConsole();

    // Call the PrintNames() function to print names to the
     ↪  console

    PrintNames(names);

    // Create a vector variable with hashes of the names using
     ↪  the CreateHashData() function

    std::vector<std::size_t> hash_table = CreateHashData(names);

    // Wait for user permission to continue

    waitForContinue();

    // Clear the console

    clearConsole();

    // Use the PrintNameHashes() function to print all names and
     ↪  hashes to the console
    PrintNameHashes(hash_table, names);

    // Wait for user permission to continue

    waitForContinue();

    // Clear the console

    clearConsole();

    // End

    return 0;

}
```

5.2 Money

```
/**
 * money.cpp
 * CS 201
 * Bryan Beus
```

```cpp
 * 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? ";

        // Initiate a while loop to wait until the user inputs a
         ↪  viable response

        while (true) {

            std::cin >> input;

            // If the response is invalid, ask again

            if (std::cin.fail() || input < 0) {
                std::cin.clear();
                std::cin.ignore(1000, '\n');
                askUserAgain();
                waitForContinue();

                // If the response is valid, input the value and
                 ↪  move to the next iteration of the for loop

            } else {
               user_wallet.push_back(input);
               std::cin.clear();
               std::cin.ignore(1000, '\n');
               break;
            }
        }
    }
}

// Calculate the wallet total as a floating point variable

float calculateWalletTotal(std::vector<int> & user_wallet) {

    float total_wallet = 0;

    // Initiate the various values of the coins

    std::vector<int> values;
        values.push_back(1);
        values.push_back(5);
        values.push_back(10);
        values.push_back(25);

        // Calcuate the total value of the wallet in pennies

    for (int i = 0; i < 4; i++) {
        total_wallet = total_wallet + user_wallet.at(i) *
         ↪  values.at(i);
    }

    // Transform the total value into a dollar value

    total_wallet = total_wallet * 0.01;

    // Return the total value

    return total_wallet;
}

// Print to the console the total wallet sum

void reportWalletSum(std::vector<int> & user_wallet,
↪   std::vector<std::string> & coin_list_plural,
↪   std::vector<std::string> & coin_list_singular) {

    clearConsole();

    // Call the calculateWalletTotal() function to calculate the
    ↪  wallet total

    float total_wallet = calculateWalletTotal(user_wallet);

    // For each coin type, print the total in the user's wallet

    for (int i = 0; i < 4; i++) {

        std::cout << "You have " << user_wallet.at(i) << " ";

        if (user_wallet.at(i) == 1) {
            std::cout << coin_list_singular.at(i);
        } else {
            std::cout << coin_list_plural.at(i);
        }

        std::cout << "." << std::endl << std::endl;

    }

    // Print the total value in the wallet

    std::cout << "The value of all your coins is $" << std::fixed
    ↪  << std::setprecision(2) << total_wallet << std::endl;

    // Pause for user to continue

    waitForContinue();

}

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

    // Clear the console

    clearConsole();

    // Declare the vector to hold the user's coin totals

    std::vector<int> user_wallet;

```

```cpp
    // Create list of plural coin names

    std::vector<std::string> coin_list_plural;
        coin_list_plural.push_back("pennies");
        coin_list_plural.push_back("nickels");
        coin_list_plural.push_back("dimes");
        coin_list_plural.push_back("quarters");

    // Create list of singular coin names

    std::vector<std::string> coin_list_singular;
        coin_list_singular.push_back("penny");
        coin_list_singular.push_back("nickel");
        coin_list_singular.push_back("dime");
        coin_list_singular.push_back("quarter");

        // Query the user's wallet

    queryUserWallet(user_wallet, coin_list_plural);

    // Clear the console

    clearConsole();

    // Report the total value

    reportWalletSum(user_wallet, coin_list_plural,
     ↪  coin_list_singular);

    return 0;
}
```

5.3 Rice

```cpp
/**
 * rice.cpp
 * CS 201
 * Bryan Beus
 * September 15, 2019
 * A program to display the power of compound interest and to
 ↪  observe the output in various variable types
 */

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

// Define the Width of the blank spaces in the cells
// This must be an even number

#define Width 6
```

```cpp

// Clear the console

void clearConsole();

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

void waitForContinue();

// Set a default function to print a series of blank spaces of
↪  length <Width>

void print_full_width(int longest_length, int col_type);

// Set a default function to print a series of blank spaces of
↪  half of length <Width>

void print_half_width();

// Set a default function to print a series of double bars of
↪  length <Width>

void print_full_bar(int longest_length, int col_type);

// Print the top of the grid

void print_top_line(int longest_length);

// Fill a whole row that has no variables or grid corners

void print_fill_row(int longest_length);

// Fill a row that has variables, including row numbers and
↪  variables inside the grid boxes
// Row requires both the current row to print and a vector that
↪  has the current state of grid boxes (X's or .'s)

void printSquare(int & currentSquare);

// Print a row that will have at least one variable value on it.

void print_var_row(int & currentSquare, int & longest_length,
↪  std::vector<std::string> & current_total_string);

// Print the bottom line of the grid

void print_bottom_line(int & longest_length);

// Print the current square

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

// Calculate new values for each important value

```

```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

        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);

        // Increase our total square count

        ++currentSquare;
    }

    return 0;
}

// Clear the console

void clearConsole() {

    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();
}

// Set a default function to print a series of blank spaces of length <Width>

void print_full_width(int longest_length, int col_type) {

    // If the column is on the left, print a Width-wide row of blank spaces

    if (col_type == 0) {
        for (int i = 0; i < Width; i++) {
            std::cout << " ";
        }

    // 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

    } else if (col_type == 1)
       for (int i = 0; i < longest_length + (Width * 2 / 3); i++) {
            std::cout << " ";
    }
}

```

```cpp
// Set a default function to print a series of blank spaces of half of length <Width>

void print_half_width() {
    for (int j = 0; j < (Width * 1 / 3); j++) {
        std::cout << " ";
    }
}

// Set a default function to print a series of double bars of length <Width>

void print_full_bar(int longest_length, int col_type) {

    // If the column is on the left, print a bar of Width length

    if (col_type == 0) {
        for (int i = 0; i < Width; i++) {
            std::cout << "";
        }

    // If the column is on the right, print a bar of a length appropriate for the longest number of grains of rice

    } else if (col_type == 1)
        for (int i = 0; i < longest_length + (Width * 2 / 3); i++) {
            std::cout << "";
    }
}

// Print the top of the grid

void print_top_line(int longest_length) {

    // Vertically clear at least one line in the terminal, then print the <Width> blank spaces

    std::cout << std::endl;

    // Print the top row of the grid

    std::cout << "   ";

    print_full_bar(longest_length, 0);

    std::cout << "";

    print_full_bar(longest_length, 1);

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

// Fill a whole row that has no variables or grid corners
```

```

void print_fill_row(int longest_length) {

    // Print a divider bar with some formatting spaces

    std::cout << "   ";

    // Call the print_full_width() function to print the left
    ↪  column

    print_full_width(longest_length, 0);

    // Print a divider bar

    std::cout << "";

    // Call the print_full_width() function to print the right
    ↪  column

    print_full_width(longest_length, 1);

    // Print a divider bar

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

}

// Fill a row that has variables, including row numbers and
↪  variables inside the grid boxes

void printSquare(int & currentSquare) {

    // Call default function to print half width of spaces

    print_half_width();

    // If the current square number is less than 10, add an extra
    ↪  space for formatting

    if (currentSquare < 10) {
        std::cout << " ";
    }

    // Print the current square number

    std::cout << currentSquare;

    // Call default function to print half width of spaces

    print_half_width();


}

```

```cpp
// Print a row in the rice/square GUI element that has variables on it

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

    // Iterate through each of the rows

    for (int i = 0; i < 3; i++) {

        // Print the first divider bar

        std::cout << "  ";

        // If this is the second row, print the square number

        if (i == 1) {
            printSquare(currentSquare);

        // Otherwise, keep the first column blank

        } else {
            print_full_width(longest_length, 0);
        }

        // Divider bar

        std::cout << "";

        // Print a bit of extra space for formatting, before printing rice grain numbers

        print_half_width();

        // Check how many blank spaces are needed to keep the current number in sync with the format of the grid

        int num_spaces = longest_length - current_total_string.at(i).length();

        // Print the number of grains of rice, and the necessary blank spaces for formatting
        std::cout << current_total_string.at(i) << std::string(num_spaces, ' ');

        // Print some more padding

        print_half_width();

        // Final divider bar

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

```

```cpp
}

// 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

    current_total_string.clear();

    // Add in the new numbers as strings

    current_total_string.push_back(std::to_string(total_in_int));
    current_total_string.push_back(std::to_string(total_in_double));
    current_total_string.push_back(std::to_string(total_in_long));

}

// Print the current measurements that answer 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) {

    // 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
    // If we have, add this value to our vector that tracks the square on which this event occurs

    if (full_total_in_int >= 1000 && values_met_int.at(0) == 0) {

        values_met_int.at(0) = currentSquare;
    }

    if (full_total_in_int >= 1000000 && values_met_int.at(1) == 0) {
        values_met_int.at(1) = currentSquare;
    }

    if (full_total_in_int >= 1000000000 && values_met_int.at(2) == 0) {
        values_met_int.at(2) = currentSquare;
    }

    if (full_total_in_double >= 1000 && values_met_double.at(0) == 0) {

        values_met_double.at(0) = currentSquare;
    }

    if (full_total_in_double >= 1000000 && values_met_double.at(1) == 0) {
        values_met_double.at(1) = currentSquare;
    }

    if (full_total_in_double >= 1000000000 && values_met_double.at(2) == 0) {
        values_met_double.at(2) = currentSquare;
    }

    // Add an extra space for formatting

    std::cout << std::endl;

    // Print the running total of grains of rice, according to the int data type

        std::cout << "Int: Total Grains of Rice Collected: " << full_total_in_int << std::endl;

    // For the int data type, for each of the three standards we measure, when they occur print them to the console

    if (values_met_int.at(0) > 0) {
        std::cout << "Int: 1000 Grains of Rice Reached on Square: " << values_met_int.at(0) << std::endl;
    }

    if (values_met_int.at(1) > 0) {
        std::cout << "Int: 1000000 Grains of Rice Reached on Square:       " << values_met_int.at(1) << std::endl;
    }

    if (values_met_int.at(2) > 0) {
        std::cout << "Int: 1000000000 Grains of Rice Reached on Square:    " << values_met_int.at(2) << std::endl << std::endl;
    }

    // Add an extra space for formatting

    std::cout << std::endl;

    // Print the running total of grains of rice, according to the double data type

        std::cout << "Double: Total Grains of Rice Collected: " << full_total_in_double << std::endl;

    // For the double data type, for each of the three standards we measure, when they occur print them to the console

    if (values_met_double.at(0) > 0) {
        std::cout << "Double: 1000 Grains of Rice Reached on Square:       " << values_met_double.at(0) << std::endl;
    }

    if (values_met_double.at(1) > 0) {
        std::cout << "Double: 1000000 Grains of Rice Reached on
         ↪  Square:    " << values_met_double.at(1) << std::endl;
    }

    if (values_met_double.at(2) > 0) {
        std::cout << "Double: 1000000000 Grains of Rice Reached
         ↪  on Square: " << values_met_double.at(2) << std::endl
         ↪  << std::endl;
    }

    // Add an extra space for formatting

    std::cout << std::endl;

    // Print the running total in the long data type

        std::cout << "Long: Total Grains of Rice Collected:
         ↪  " << full_total_in_long << std::endl;

    // When we reach the end of all calculations, print our
     ↪  result in the console

    if (currentSquare == 64) {
        std::cout << "The above value is the total of all grains
         ↪  of rice on all squares." << std::endl;
    }

    // Add an extra space for formatting

    std::cout << std::endl;

    // Calculate the square on which the int or the double data
     ↪  type may stop keeping up with our running total

    if (square_int_tripped == 0 && total_in_int <= 0) {
        square_int_tripped = currentSquare;
    }

    if (square_double_tripped == 0 && total_in_double <= 0) {
        square_double_tripped = currentSquare;
    }

    // Report the square on which any failed data type
     ↪  experienced the failure

    if (square_int_tripped != 0) {

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

    if (square_double_tripped != 0) {

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

}
```

5.4 Scores

```
/**
 * scores.cpp
 * CS 201
 * Bryan Beus
 * September 18, 2019
 * A program to record names and scores in two separate vectors
 */

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

using std::cout;
using std::endl;
using std::string;
using std::vector;
using std::cin;

// Clear the console

void clearConsole();

// Wait for user input

void waitForContinue();

// Display the main prompt screen

bool displayPrompt(vector<string> & names, vector<int> & scores);

// Request the user to input a name and score

bool addInput(vector<string> & names, vector<int> & scores);

// Request a name

bool requestName(vector<string> & names, string & newName);

// Request a score

```

```cpp
bool requestScore(vector<int> & scores, int & newScore, string & newName);

// Check that the inputted name is not a duplicate

bool checkOriginal(string & newName, vector<string> & names);

// Check that the lengths of the vectors match

bool checkLengths(vector<string> & names, vector<int> & scores);

// Check that the database is not empty

bool checkDatabaseHasInputs(vector<string> & names);

// Print a list of names and scores

bool printList(vector<string> & names, vector<int> & scores, bool wait);

// Search for a name in the database

void searchName(vector<string> & names, vector<int> & scores);

// Search for a score in the database

void searchScore(vector<string> & names, vector<int> & scores);

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

  // Declare names and scores vectors

 vector<string> names;
 vector<int> scores;

  // Initiate endless while loop to maintain program stream

 while (true) {

    // Initiate bool variable to allow program to end, if needed
    // Call main displayPrompt() function to beging program

  bool result = displayPrompt(names, scores);

    // If result is ever returned negative, end the program

  if (!result) {
   break;
  }
 }

 return 0;
}

```

```cpp
// Clear the console

void clearConsole() {
  cout << "\033[2J\033[1;1H";
}

// Wait for user input

void waitForContinue() {
  cout << "Press enter to continue . . . ";
  getchar();

    // Clear cin

    cin.clear();
    cin.ignore(1000, '\n');
}

// Display the main prompt screen

bool displayPrompt(vector<string> & names, vector<int> & scores)
 ↪  {

    // Clear the console

  clearConsole();

    // Declare a result variable that will be sent back to the
     ↪  int main() function's while loop

  bool result = true;

    // Declare option variable to capture user input for program
     ↪  direction

    int option;

    // Display options

  cout << "Choose an option from the following menu: " << endl;
  cout << "\n1) Add new names and scores to the database" << endl;
  cout << "2) Print the full list of names and scores" << endl;
  cout << "3) Search for a name" << endl;
  cout << "4) Search for a score" << endl;
  cout << "0) End program\n" << endl;

    // Initiate endless while loop
    // Loop continues until user provides valid input

    while (true) {

        // Capture input

        cin >> option;

        // Ensure input is valid

        if (cin.fail() || option > 4 || option < 0) {

            // Clear cin

            cin.clear();
            cin.ignore(1000, '\n');
            cout << "\nThe option you selected is not valid.
            ↪  Please try again: ";

            // If the input is valid, then break the loop to
            ↪  continue

        } else {
            break;
        }
    }

    // Initiate switch method to determine response to input

  switch (option) {

        // Add an input

    case 1:
      result = addInput(names, scores);
      break;

        // Print a list of names

    case 2:
      printList(names, scores, true);
      break;

        // Search for a name in the database

    case 3:
      searchName(names, scores);
      break;

        // Search for a score in the database

    case 4:
      searchScore(names, scores);
      break;

        // End the program by returning false

    case 0:
      result = false;
      break;

        // Restart stream by default

    default:
      break;
  }

    // Return the result to the endless while loop in int main()

  return result;
 }

 // Request the user to input a name and score

 bool addInput(vector<string> & names, vector<int> & scores) {

    // Declare result variable to monitor user progress, and if
    ↪  necessary, return a false value, and thus end the program

    bool result = true;

    // Clear the console

  clearConsole();

    // Print request for user input

  cout << "Please enter names and scores for the database." <<
  ↪  endl;
  cout << "\nTo indicate that you are finished entering data,
      ↪  input a name as \"NoName\" paired with a score of \"0\"\n"
      ↪  << endl;

    // Initiate endless while loop to request new name

  while (true) {

        // Declare newName variable
        // Declare newScore variable

        string newName;
        int newScore;

        // Declare result variable to ensure call to
        ↪  requestName() function is succesfull
        // Call requestName() function

        result = requestName(names, newName);

        // If the result is unsuccessful, end the loop (and
        ↪  program)

        if (!result) {
            break;

            // Otherwise, call the requestScore() function and set
            ↪  the response to the result variable
```

```

        } else if (result) {
            result = requestScore(scores, newScore, newName);
        }

        // Check to see if the input values are NoName and 0
        // If so, end the while loop

        if (newName == "NoName" && newScore == 0) {
            break;
        }
    }

    // Check that the length of the names and scores vectors are
    ↪  the same
    // If not, end the program with error message

    if (!checkLengths(names, scores)) {
        cout << "The names and scores vectors are of different
        ↪  lengths. Something is wrong in the code." << endl;
        result = false;
    }

    // Return the result

    return result;
}

// Request a name for the names vector

bool requestName(vector<string> & names, string & newName) {

    // Declare the result variable to return at the end

    bool result = true;

    // Print request for new name

    cout << "Enter a new name for the database: ";

    // Initiate endless while loop to request new name

    while (true) {

        // Capture new name

        cin >> newName;

        // If the input type is invalid, start again

        if (cin.fail()) {
            cin.clear();
            cin.ignore(1000, '\n');
           cout << "\nThe input you provided is not valid. Please
            ↪  try again.\n" << endl;

            // Check that this input name is original
            // If it is not, end the program

        } else if (!checkOriginal(newName, names)) {
            cin.clear();
            cin.ignore(1000, '\n');
            cout << "\nThe name you provided is already in the
             ↪ database. Terminating program (as per
             ↪ instructions)." << endl;
            result = false;
            break;

            // Check to see if the user is beginning the
             ↪ termination process

        } else if (newName == "NoName") {
            break;

            // Add the name to the database
        } else {
            names.push_back(newName);
            break;
        }
    }

    // Return the result

    return result;
}

// Request a new score

bool requestScore(vector<int> & scores, int & newScore, string &
 ↪  newName) {

    // Declare result variable to monitor function progress

    bool result = true;

    // Check to see if the current newName variable is NoName
    // If it is, request user to confirm by entering 0
    if (newName == "NoName") {


        cout << "\nPlease confirm by entering '0' as a score.\n"
         ↪  << endl;

        // Initiate endless while loop to capture valid user
         ↪ response

        while (true) {

            // Capture user input

            cin >> newScore;

            // Verify user input

            if (cin.fail()) {
                cin.clear();
                cin.ignore(1000, '\n');
                cout << "The input provided is not a valid
                ↪  integer. Please try again." << endl;

                // If the user input is 0, return to the main
                ↪  display prompt

            } else if (newScore == 0) {
                return result;

                // If the user enters a valid input other that 0,
                ↪  continue putting new names and scores into
                ↪  the database

            } else {
               cout << "\nContinuing with name and score database
                ↪  inputs.\n" << endl;
                break;
            }
        }

    // If the newName variable is not 'NoName,' begin the process
    ↪  for collecting a matching score

    } else {

        // Request new score

        cout << "Enter the score for " << newName << ": ";

       // Initiate endless while loop to capture valid user score

        while (true) {

            // Capture new score value

            cin >> newScore;

            // If input is invalid restart the loop

            if (cin.fail()) {

                cin.clear();
                cin.ignore(1000, '\n');
                cout << "The input provided is not valid. Please
                ↪  try again: ";

            // Otherwise, add the new score to the database and
             ↪ end the loop

            } else {

                scores.push_back(newScore);
                break;

            }
        }
    }

    // Return the result variable

    return result;

}

// Check that the input newName variable is not a duplciate

bool checkOriginal(string & newName, vector<string> & names) {

    // Declare isOriginal variable to check whether newName is new

    bool isOriginal = true;

    // Iterate through names vector to check for duplicates

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

        // If a duplicate is found, set isOriginal to false
        if (names.at(i) == newName) {
            isOriginal = false;
        }
    }

    // Return result

    return isOriginal;
}

// Check to make sure that names and scores vectors are valid

bool checkLengths(vector<string> & names, vector<int> & scores) {

    // Check that the lengths are correct, and return a bool
     ↪ result

    bool isCorrect = (names.size() == scores.size()) ? true :
     ↪ false;

    // Return the bool result

    return isCorrect;

```

```
}

// Check that the database of names is not empty

bool checkDatabaseHasInputs(vector<string> & names) {

    // If size of names vector is less than one, return
    ↪  instructions to user, wait for user to confirm, and then
    ↪  end the function with a negative

    if (names.size() < 1) {
        cout << "You must put names and scores in the database
        ↪  before attempting to read it." << endl;
        waitForContinue();
        return false;
    }

    // If the size is greater than or equal to one, return
    ↪  positive

    return true;

}

// Print a list of provided names and scores

bool printList(vector<string> & names, vector<int> & scores, bool
↪  wait) {

    // Clear the console

    clearConsole();

    // Check that the database is not empty

    if (checkDatabaseHasInputs(names)) {

        // Print the names and scores in columns

        while (true) {

            // Declare names and scores title strings

            string columnOne = "Names";
            string columnTwo = "Scores";

            // Declare variable for standard char size of column
            ↪  one

            int columnSize = columnOne.length() + 3;

            // Iterate through list of names and discover longest
            ↪  column one name

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

```cpp
                if (names.at(i).length() > columnSize + 3) {
                    columnSize = names.at(i).length() + 3;
                }
            }

            // Declare variables to manage column formatting

            int columnTitleSpaces = columnSize -
             ↪  columnOne.length();
            int dashCount = columnTwo.length() + columnSize;

            // Print column titles

            cout << columnOne <<
             ↪  string(std::max(columnTitleSpaces, 3), ' ') <<
             ↪  columnTwo << endl;
            cout << string(dashCount, '-') << endl;

            // Print all names and scores

            for (int i = 0; i < names.size(); i++) {
                cout << names.at(i) << string(columnSize -
                 ↪  names.at(i).length(), ' ') << scores.at(i) <<
                 ↪  endl;
            }

            // Wait for user to continue

            if (wait) {
                waitForContinue();
            }

            // End while loop

            break;

        }
    }
}

// Search for a name in the database

void searchName(vector<string> & names, vector<int> & scores) {

    // Clear the console

    clearConsole();

    // Declare searchName variable
    //  Initiate nameFound variable and begin with default at
     ↪  negative

    string searchName;
    bool nameFound = false;

    // Declare a tempName and tempScore varaible
    // If name(s) and score(s) are found, these variable are sent
     ↪  to the printList() function

    vector<string> tempName;
    vector<int> tempScore;

    // Verify that the database is not empty

    if (checkDatabaseHasInputs(names)) {

    // Print request for user input

    cout << "Enter the name for which to search the database: ";

    // Initiate endless while loop to request valid user input

        while (true) {

            // Capture user input

            cin >> searchName;

            // If input is invalid, restart loop

            if (cin.fail()) {
                cin.clear();
                cin.ignore(1000, '\n');
                cout << "The input you provided is not valid.
                 ↪  Please try again: ";

            // Iterate through the list of names and search for
             ↪  matching names

            } else {
                for (int i = 0; i < names.size(); i++) {
                    if (names.at(i) == searchName && !nameFound) {

                        // If name is found, set nameFound to true

                         nameFound = true;

                         // Set tempName and tempScore values and
                          ↪  send call to printList() function

                         tempName.push_back(names.at(i));
                         tempScore.push_back(scores.at(i));
                         printList(tempName, tempScore, true);

                    }
                }

                // If no matching name is found, inform the user

                if (!nameFound) {
                    cout << "\nName not found." << endl;
                    waitForContinue();
                }

                // End loop

                break;
            }
        }
    }
}

// Search database for a score

void searchScore(vector<string> & names, vector<int> & scores) {

    // Clear the console

    clearConsole();

    // Declare searchScore and scoreFound variables

    int searchScore;
    bool scoreFound = false;

    // Declare tempName and tempScore variables to hold potential
    ↪  name(s) and score(s)

    vector<string> tempName;
    vector<int> tempScore;

    // Check that the database is not empty

    if (checkDatabaseHasInputs(names)) {

        // Request user input

        cout << "Enter the score for which to search the database:
        ↪  ";

        // Initiate endless while loop to request valid user input

        while (true) {

            // Capture user input

            cin >> searchScore;

            // If input is invalid, restart loop

            if (cin.fail()) {
                cin.clear();
                cin.ignore(1000, '\n');
```

```cpp
                cout << "The input you provided is not valid. Please try again: ";

            // Iterate through the list of scores and capture all matching values
            // For each matching value, push to the tempName and tempScore vectors

            } else {
                for (int i = 0; i < scores.size(); i++) {
                    if (scores.at(i) == searchScore) {
                        scoreFound = true;
                        tempName.push_back(names.at(i));
                        tempScore.push_back(scores.at(i));

                        // Send all matching names and scores to printList() function for printing

                        printList(tempName, tempScore, false);
                    }
                }

                // If no matching score is found, inform the user

                if (!scoreFound) {
                    cout << "\nScore not found." << endl;
                }

                // Wait for user to indicate readiness

                waitForContinue();

                // End loop

                break;
            }
        }
    }
}
```