Mastermind

version 1.1

Stephanie Peacock

CIS-7

Fall 2024

Description....3
GitHub....3
https://github.com/StephaniePeacock/CIS7....3
Gameplay....3
Main:....3
Prompt:....3
Code Generation:....4
Making a Guess:....4
Checking the Guess against the Code:....4
Outputting the Result:....5
Result Example:....5
Flowcharts....6
Main:....6
Make(guess):....7
Check:....8
Result:....9

Description

A code breaking game. The user is given a finite number of attempts to properly guess the randomized code. Feedback is given after each guess to allow the user to modify their guess.

GitHub

https://github.com/StephaniePeacock/CIS7

Gameplay

Main:

The main function sets the random seed. Several variables are declared. Size, Choices, Tries are left unset, as the user will enter them in the prompt function. Total is set to 0.

The user is prompted to enter the game settings. Arrays sized to the user's choice are declared for both the code and the guess. A bool match is set initially to false.

The program generates a code, then begins a loop, which runs until tries is 0. Each iteration of the loop asks the user to make a guess. Match's truth value is set to the check function's return value. The number of tries is decremented, and the total number of attempts is incremented. If match returned true, then the user has entered the correct code, so tries is set to 0, ending the loop early.

The game ends when the number of attempts has been exhausted. The program then checks to see if all values were matched. If there weren't, then the user is notified the code was unbroken. If all values were matched, the user is notified how many attempts it took them to break the code.

Prompt:

At the beginning of the prompt, the user is asked to set the options for the game. They will choose the length of the code to break, the range of digits that are possibilities, and the number of attempts they wish to have to break the code. Two separate counters are also implemented, one to keep track of the number of attempts remaining, and one to keep track of the number of attempts used.

Code Generation:

The program randomizes a code that matches the length the user selected, limiting each digit to be within range of the user's choice. The range will always include 0. If the user selects to have 5 possible digits, the program will choose either a 0,1,2,3,4 for each element of the code array.

Following code generation, the gameplay loop begins.

Making a Guess:

After the code has been created, the user is prompted to make a guess. Checks are in place to ensure that the guess has the correct number of characters, and that each character entered is a digit. If either the length of the guess, or any character entered is not a valid digit, the user is prompted to reenter their guess.

Checking the Guess against the Code:

The check function returns a bool, match, which is initially set to true. The user's entered guess is compared to the code. This is accomplished with a series of loops. The first loop checks if any digits from the guess are an exact match for both the digit and the placement. If there are, then the exact[] array is incremented for that element. If any of the digits are not an exact match, then match is set to false.

To prevent double counting of partial matches, the second loop counts the number of times each digit is present in the code, using a nested loop to compare each element to each other element.

Finally, partial matches are checked for using another nested loop. Within the outer loop, we first check if that element is an exact match. If it is, then the outer loop increments. If that element is not an exact match, the inner loop begins.

The first check is against the count[n], to see if this element has already been counted. If it has, (and count[n] is now 0) then the inner loop continues, iterating to the next inner loop element. If the count[n] is 1 or more, and that exact[n] is false, then guess[i] is checked against code[n]. If there is a match, partial[i] is marked true, count[n] is decremented, and n is set to size, to skip to the next code[n].

After all checks have been completed, the result is printed for the user to evaluate, and the value of match is returned.

Outputting the Result:

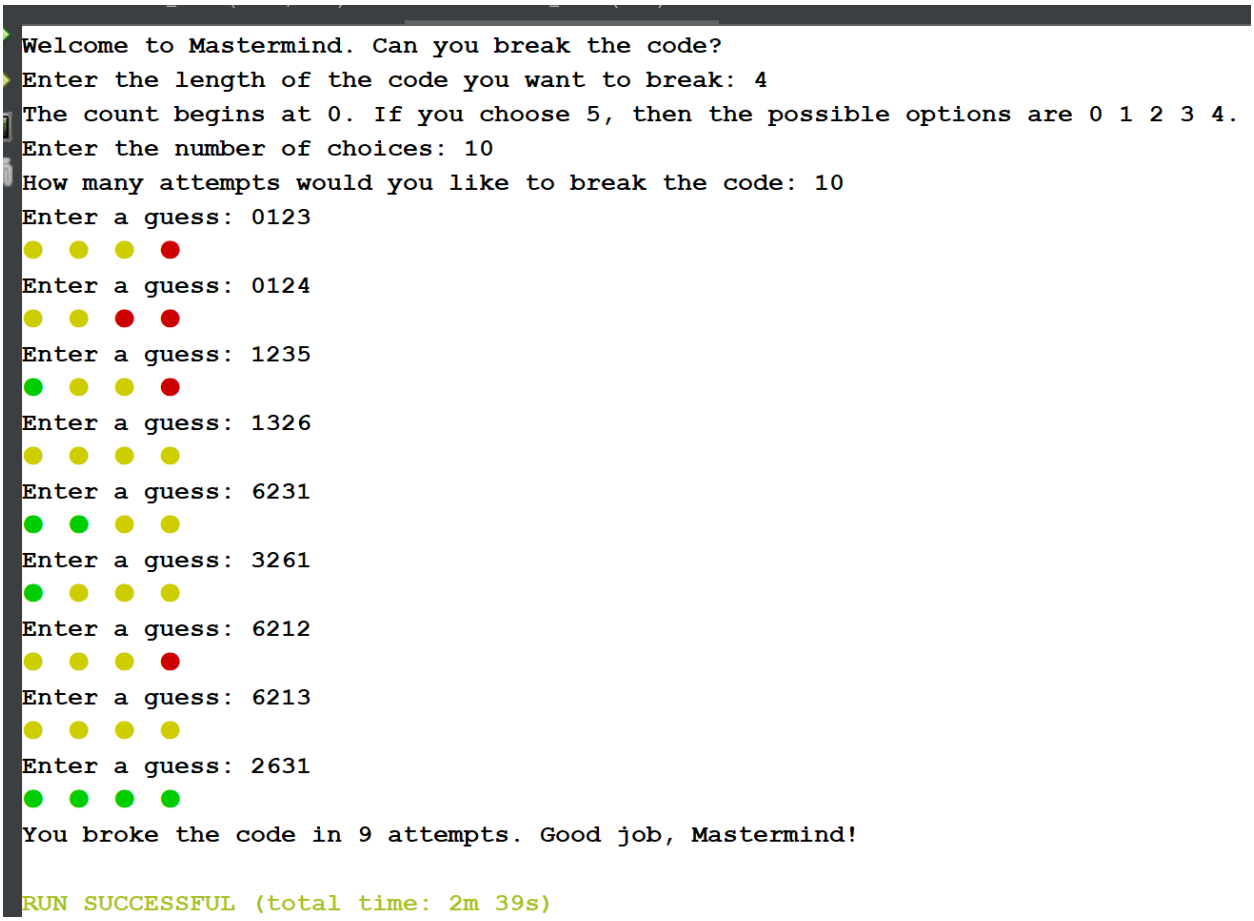
The result function is called within check. I wanted to make it look visually appealing, so I created string variables to change the color, as well as the Unicode for a bullet point. I obtained these values through a google search (must as I did with Suit symbols for my precious card games).

In the traditional game, Mastermind does not order the feedback based on placement within the code. Instead, feedback is given in the following order:

- Number of correct digits in the correct spot
- Number of correct digits in the wrong spot
- Number of digits that are not in the code at all

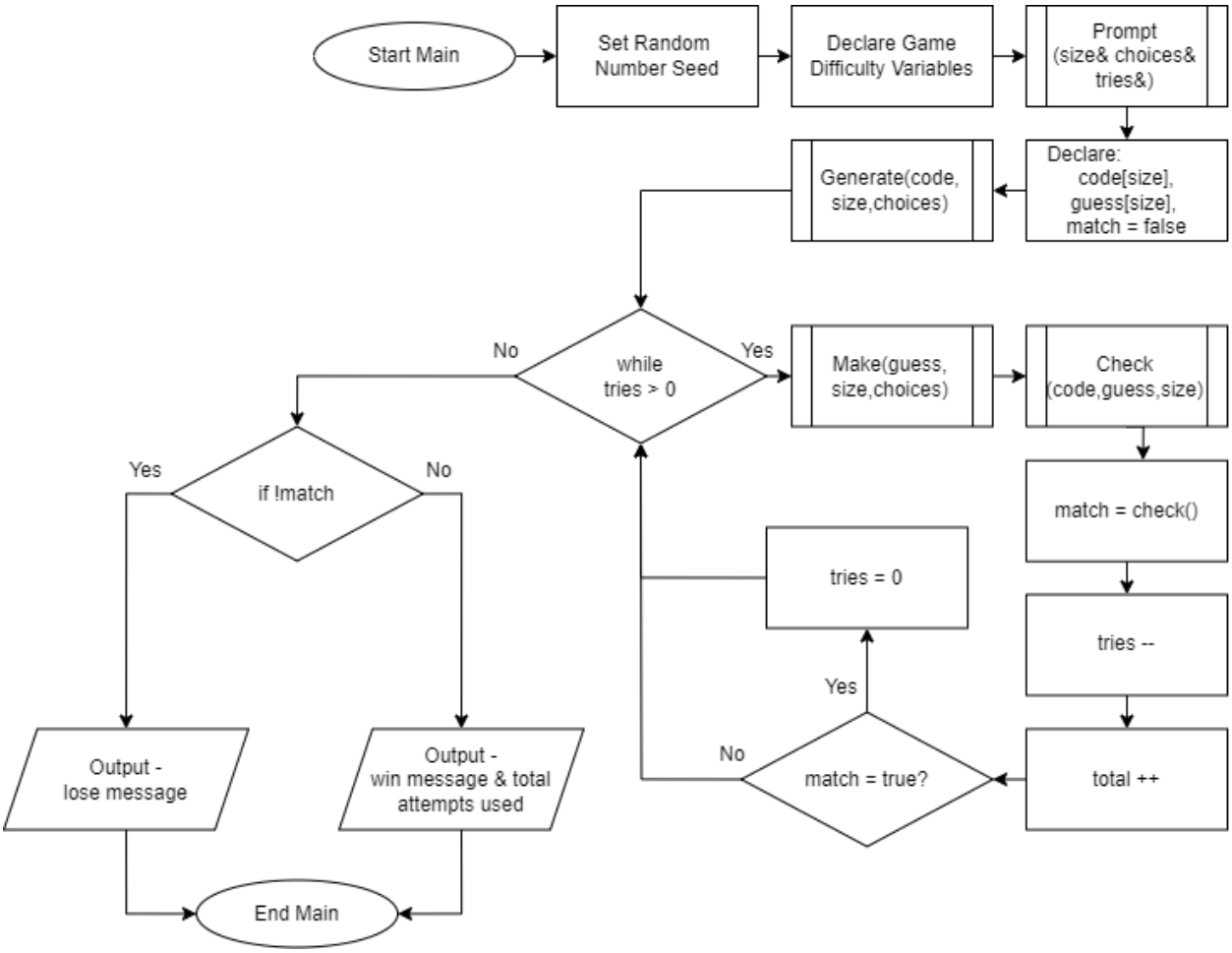
This order was accomplished using three simple loops. Each loop first sets the color, then outputs the dot, followed by a space. After all three loops have been processed, the black color is output, so the console color is reset to black for the next round.

Result Example:



Flowcharts

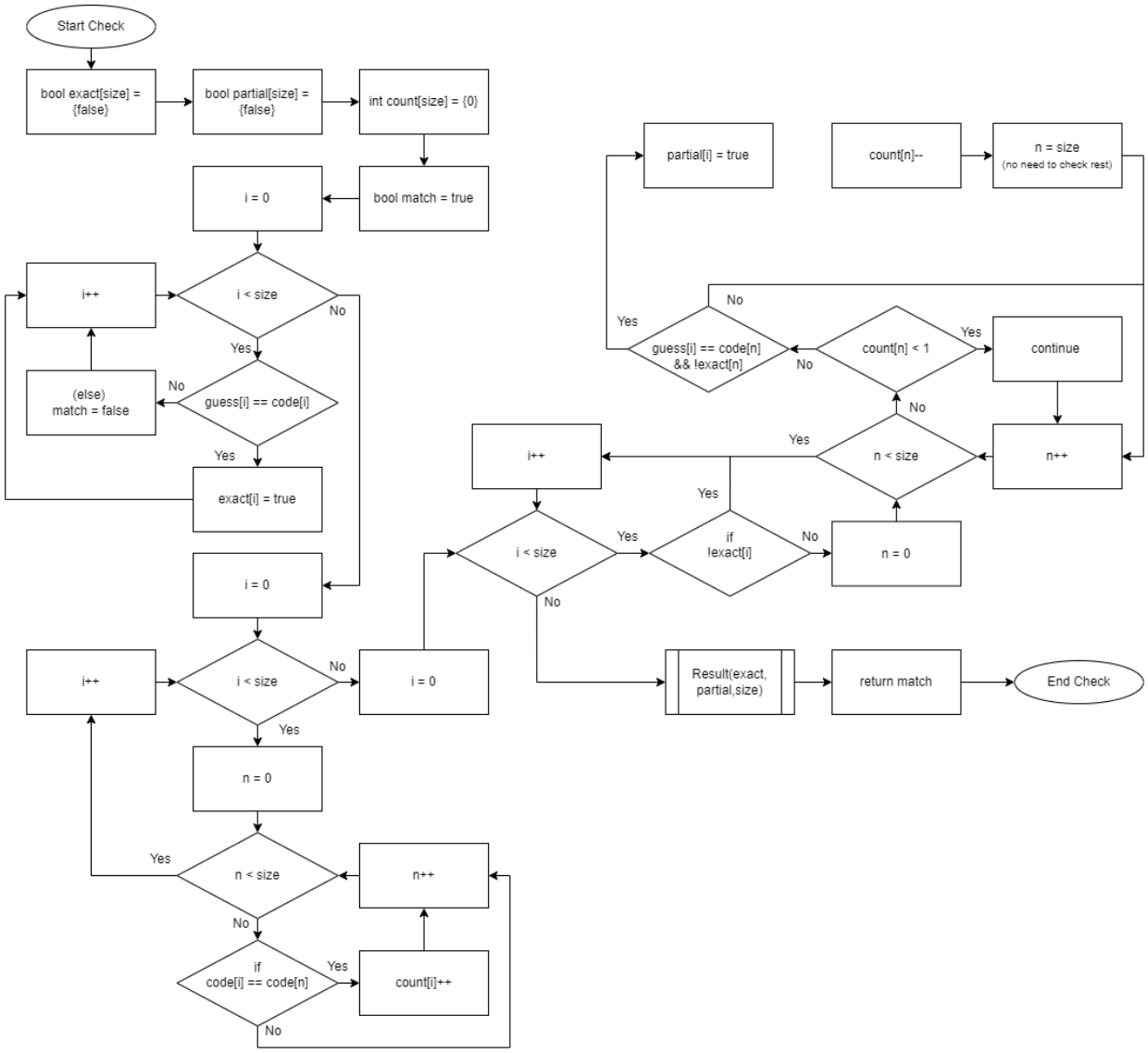
Main:



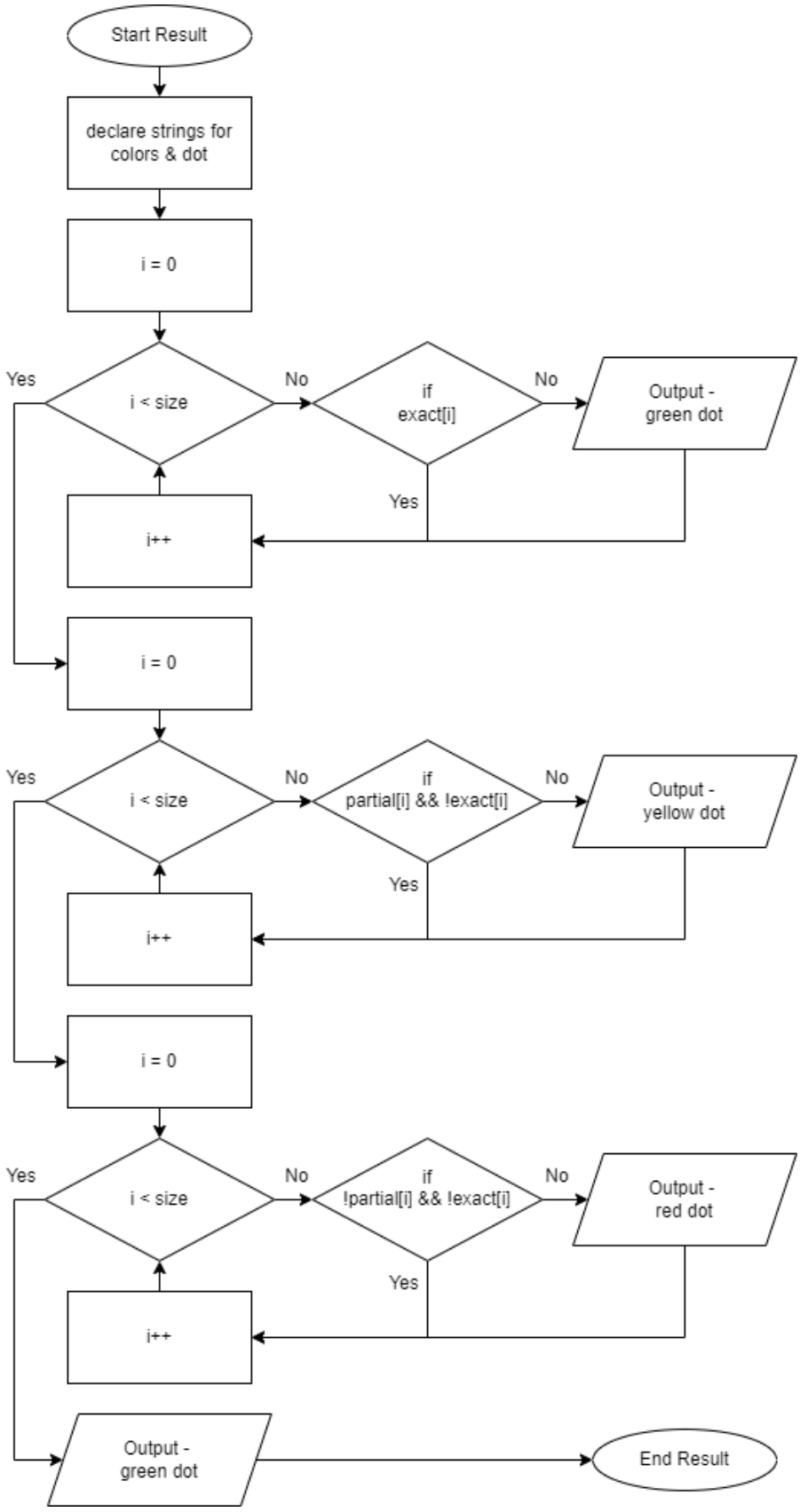
Make(guess):


Start Make
Declare:
string input
bool invalid = true
Output -
ask for guess
while
invalid
Yes
No
End Make
getline(cin, input)
if
input.length() == size
No
else Output -
wrong size
Yes
invalid = false
invalid = true
i = 0
i++
i < size
No
Yes
guess[i] = input[i] + '0'
Yes
if input[i]
is digit & in range
No
i = size
invalid = true
else Output -
not in range

Check:


Start Check
bool exact[size] = {false}
bool partial[size] = {false}
int count[size] = {0}
bool match = true
i = 0
i++
i < size
No
Yes
(else) match = false
No
guess[i] == code[i]
Yes
exact[i] = true
i = 0
i++
i < size
No
Yes
n = 0
Yes
n < size
n++
No
if code[i] == code[n]
Yes
count[i]++
No
i = 0
i++
i < size
No
Yes
if !exact[i]
Yes
No
n = 0
No
Result(exact, partial,size)
return match
End Check
partial[i] = true
count[n]--
n = size
(no need to check rest)
No
Yes
guess[i] == code[n] && !exact[n]
No
count[n] < 1
Yes
continue
No
Yes
n < size
n++

Result:


Start Result
declare strings for colors & dot
i = 0
Yes
i < size
No
if exact[i]
No
Output - green dot
Yes
i++
i = 0
Yes
i < size
No
if partial[i] && !exact[i]
No
Output - yellow dot
Yes
i++
i = 0
Yes
i < size
No
if !partial[i] && !exact[i]
No
Output - red dot
Yes
i++
Output - green dot
End Result